Enabling End-host Network Functions

Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P. Grosvenor,*
Thomas Karagiannis, Lazaros Koromilas,† and Greg O'Shea
Microsoft Research
Cambridge, UK

ABSTRACT

Many network functions executed in modern datacenters, *e.g.*, load balancing, application-level QoS, and congestion control, exhibit three common properties at the data plane: they need to access and modify state, to perform computations, and to access application semantics — this is critical since many network functions are best expressed in terms of application-level messages. In this paper, we argue that the end hosts are a natural enforcement point for these functions and we present Eden, an architecture for implementing network functions at end hosts with minimal network support.

Eden comprises three components, a centralized controller, an enclave at each end host, and Eden-compliant applications called stages. To implement network functions, the controller configures stages to classify their data into messages and the enclaves to apply action functions based on a packet's class. Our Eden prototype includes enclaves implemented both in the OS kernel and on programmable NICs. Through case studies, we show how application-level classification and the ability to run actual programs on the data-path allows Eden to efficiently support a broad range of network functions at the network's edge.

CCS Concepts

•**Networks** → **Programmable networks; Network management;** *Data center networks;* Cloud computing;

*Work performed while an intern with Microsoft Research; currently at Cambridge University, UK

†Work performed while an intern with Microsoft Research; currently at University of Crete, Greece

SIGCOMM '15, August 17 - 21, 2015, London, United Kingdom
 ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787493

Keywords

Software Defined Networking; SDN; Network Management; Data-plane programming; Network Functions

1 Introduction

Recent years have seen a lot of innovation in functionality deployed across datacenter networks. *Network functions* range from management tasks like load balancing [65, 4, 26, 1, 40] and Quality of Service (QoS) [9, 39, 10, 33, 52, 61, 6, 28] to data-plane tasks like (centralized) congestion control and network scheduling [48, 27, 64, 45, 30] to application-specific tasks like replica selection [17]. Today, such functions are implemented using a mishmash of techniques and at a variety of places—at network switches using SDN and OpenFlow, at physical or virtual middleboxes using NFV, and at end hosts through Open vSwitch or custom implementations.

Despite their differences, three common requirements characterize a large fraction of network functions: i) they maintain state at the data plane, ii) they perform computation at the data plane, and iii) they operate on application semantics. The last feature is particularly important since many network functions are best expressed in terms of application data units or "messages". For example, a load balancing function for memcached, a popular key-value store, may put its messages into two "classes" (GETs and PUTs) and treat them differently. It may even use message-specific details like the key being accessed for load balancing [40].

Traditional network management approaches implement network functions completely decoupled from applications. They infer the application message a packet belongs to using deep packet inspection or through other heuristics. Instead, we propose *messages* and *classes* as first-order abstractions at the network layer. A message refers to an (arbitrary) application data unit while a class is the set of messages (and consequent network packets) to which the same network function should be applied. Applications can provide the class and message information for any traffic they generate.

In this paper, we present Eden, an architecture for implementing network functions at end hosts. End hosts are a natural enforcement point for such functions—

they have plentiful resources that allow for complex computation and large amounts of state to be maintained, and they are ideally placed for fine-grained visibility into application semantics. Finally, in single administrator environments like enterprises and datacenters, some part of end hosts can be trusted. We show that a large number of diverse and interesting network functions can be efficiently realized at the end hosts with minimal support from the network.

Eden comprises three components: a logically centralized *controller*, *stages*, and end host *enclaves*. A stage is any application or library that is Eden-compliant. Stages bridge the gap between network functions expressed in terms of application-level messages and the enclave operating on packets. To achieve this, stages *classify* their network traffic, associating application messages with a class and a message identifier that is carried with it down the host's network stack. The enclave resides along the end host network stack, either in the OS or the NIC. It extends and replaces functionality typically performed by the end host virtual switch. An enclave has a set of match-action tables that, based on a packet's class, determine an *action function* to apply. The action function can modify both the packet and the enclave's global state. Enclaves and stages each expose an API through which they can be programmed. Enclaves and stages, taken together, enable application-aware data plane programmability.

Given a network function, the controller can implement it by programming stages and enclaves across the network. Hence, Eden achieves a careful division of functionality; the controller provides global visibility, stages provide application visibility while enclaves provide a pragmatic enforcement point at the data plane.

A key challenge posed by our design is efficient and safe execution of action functions at enclaves while allowing for the functions to be dynamically updated by the controller without impacting data plane performance. With Eden, action functions are written in a high-level domain specific language using F# code quotations. They are compiled to bytecode which is then interpreted through a stack-based interpreter within the enclave. This approach allows Eden to execute the same computation across multiple platforms and avoids the complexities of dynamically loading code in the OS or the NIC. Indeed, our interpreter can execute the same action function inside the OS or in a programmable NIC.

We have implemented the Eden enclave across two platforms: Hyper-V and Netronome's programmable NICs [46]. We evaluate Eden through case studies across these platforms. These studies highlight how the Eden architecture can implement diverse functions spanning application-aware load-balancing, quality of service and weighted path selection. We show that Eden's interpreter-based data plane computation incurs reasonable overheads with negligible impact on application performance metrics.

Overall, this paper makes the following contributions:

- We highlight that a large class of network functions feature three key requirements: data-plane computation, data-plane state, and operate on application semantics (§2).
- We design and implement Eden, an architecture that enables end host network functions through data plane programmability (§3).
- We present a flexible scheme for application-level classification of network traffic (§3.3).
- We present a language, compiler and runtime for action functions. The compiler decouples state management from the function, thus providing a clean programming abstraction to administrators (§3.4).

The idea of end hosts participating in the implementation of network functions is not new [49, 16, 21, 34, 59, 57]. These state of the art approaches, however, still encourage a low-level, packet-based API for programming the data plane, often a variant of OpenFlow. This ignores end host capabilities and restricts the functions that can be implemented. Instead, Eden adopts a different philosophy by introducing a data plane interface that is wide and rich enough to allow for general, application-informed data-plane computation.

2 Network Functions

This paper is motivated by the observation that three common data-plane requirements underlie many network functions. First, they need to create, access and modify *state*, in many cases on a per-packet basis. This allows for "stateful network functions" where a packet influences the processing of subsequent packets. Second, they require *computation* on the data path.

Finally, they require visibility into *application semantics*. This introduces a mismatch – data-plane elements like the end host network stack and switches operate at the granularity of packets. Instead, each application has its own application data unit or "message". For example, for memcached, a popular key-value store, a message is a GET or a PUT request or response. For a HTTP library, a message is an HTTP request or response. These messages are fragmented into packets before being sent across the network. Many network functions however are best expressed in terms of application messages. Implementing such network functions thus requires a mapping between messages and the consequent network packets.

2.1 Examples

To make the discussion concrete, we now use a few network functions proposed in recent literature as case-studies to highlight the three requirements mentioned above. For each example, we present a pseudo-code sketch for the network function; in Section 5, we show how the functions can be expressed in Eden's language.

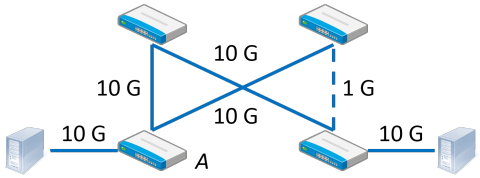

Figure 1: Example of an asymmetric topology.

2.1.1 *Load balancing*

Many recent proposals focus on load balancing of traffic across network paths and across destinations in datacenters. We begin with a simple path-level load balancing example.

Datacenters today use ECMP to load balance traffic among multiple equal cost paths. However, ECMP assumes that the underlying topology is balanced and regular which is often not true in practice [65]. For example, in Figure 1, ECMP would distribute the traffic at switch A evenly across the two upstream links even though subsequent links along the paths have very different capacities. This can result in unfairness and poor tail performance. Motivated by this, Zhou et al. [65] proposed WCMP, a weighted version of ECMP that balances traffic across links in a weighted fashion. In the example in Figure 1, WCMP can be used to balance traffic across the two upstream links of A in a 10:1 ratio. While the authors present a switch-based implementation of WCMP in [65], we focus on the higher-order network function associated with WCMP and for now, sidestep where this function is implemented.

WCMP requires two main pieces of functionality: (a) for each source-destination pair, computation of paths and weights, and (b) picking a path for packets. The former requires information about the network topology which, in turn, requires a coordination point with global visibility. With SDN-based management approaches, such functionality is typically implemented at a centralized controller, possibly using high-level programming abstractions [25, 62, 24, 54]. Here, we focus on the latter operation that needs to operate at the data path.

The first snippet in Figure 2 implements the `WCMP` data-plane function. It takes a packet as input and chooses its network path based on the weights of the links between the packet's source and destination. These weights are obtained periodically from the controller. Next, we discuss the state maintained by this function and the computation it performs.

State. The function accesses global state—it reads the `pathMatrix` variable that, for each source and destination pair, contains the list of paths between them and their weights. A path's weight is the probability with which it should be used and is a product of the normalized weights of all links along the path.

```
Require: Global program state- pathMatrix, which
    gives the list of paths and their weights for each
    source destination pair.
    pathMatrix:[src, dst] -> {[Path1, Weight1],
    ...}

1: fun WCMP (packet) {
2:   Choose a path in a weighted random fashion from
     pathMatrix[p.src, p.dst])

3:   Send packet
4: }
```

```
Require: Each packet is tagged with its message iden-
    tifier

1: fun messageWCMP (packet) {
2: msg = packet.message
3: if packet belongs to new message then
4:     Choose a path in a weighted random fashion from
       pathMatrix[p.src, p.dst]

5:     cachedPaths[msg] = path

6: Use cachedPaths[msg] as this packet's path
7: Send packet
8: }
```

Figure 2: Network functions implementing weighted load balancing at packet- and message granularity.

Computation. The function performs a simple computation — it chooses the packet's path from the set of possible paths in a weighted random fashion.

Application semantics. The `WCMP` function chooses paths on a packet-by-packet basis. However, as we show in Section 5, this can lead to packet re-ordering across the network which can adversely impact the performance of connection oriented protocols like TCP. Flow-level weighted load balancing [65] can address this problem, at the expense of some load imbalance.

More broadly, datacenter operators can balance the trade-off between application performance and load across network links through "*message-level load balancing*" whereby all packets of the same application message are sent along the same path. For example, for memcached traffic, all packets corresponding to a GET or a PUT operation are sent along the same path. This is implemented by the `messageWCMP` function in Figure 2 which needs to know the message a given packet belongs to.

The previous discussion also applies to other schemes for load balancing in datacenters [4, 26, 1]. For example, Ananta [4] load balances incoming connections across a pool of application servers. This requires a NAT-like functionality along the network path which, in turn, requires both computation and state. Several proposals explicitly use application semantics as

```
Require: Each packet is tagged with its message, size
    and tenant information.
    Global state: queueMap, a map that returns the rate
    limited queue for each tenant

 1: fun Pulsar (packet) {
 2: msg = packet.message
 3: if msg is a "READ" message then
 4:     Set size to msg.size //if read, policing is based
        on operation size
 5: else
 6:     Set size to packet.size //policing is based on the
        packet size
 7: Send packet to queue queueMap[packet.tenant] and
    charge it size bytes
 8: }
```

Figure 3: Pulsar's rate control function

```
Require: Each packet is tagged with its message and
    size information.
    Global state: msgTable:[msg] -> bytesSent and
    priorityThresholds

 1: fun PIAS (packet) {
 2: msg = packet.message
 3: if packet belongs to a new message then
 4:     Initialize msgTable[msg] to packet.size
 5: else
 6:     Increment msgTable[msg] by packet.size
 7: Set packet priority according to priorityThresh-
    olds
 8: Send packet
 9: }
```

Figure 4: Network function implementing a generalized version of PIAS' dynamic priority logic.

well. For example, Sinbad [17] maximizes performance by choosing endpoints for write operations in HDFS, and Facebook's mcrouter [40] achieves low latency and failover through a distributed middle layer that routes memcached requests based on their key.

2.1.2 Datacenter QoS

Many recent proposals provide quality of service (QoS) guarantees for traffic across the datacenter network. We begin with a recent QoS solution.

Pulsar [6] gives tenant-level end-to-end bandwidth guarantees across multiple resources, such as network and storage. A tenant refers to a collection of virtual machines (VMs) owned by the same user. Enforcing such guarantees at the network layer poses two key challenges. First, it requires determining the tenant a given packet belongs to. Second, since the guarantee spans non-network resources, it requires knowledge of the operation being performed. For example, consider a packet corresponding to a request to a storage server. Network rate limiters pace packets based on their size. However, depending on whether the packet corresponds to a READ or a WRITE request to a storage server, the packet size may not correctly represent its impact on the storage server—read packets are small on the forward path from the client generating the request to the server, but their cost on the server and the reverse network path can be high depending on the read size. Today, information about the type of an operation and its size is not available to network rate limiters at end hosts and at switches.

Figure 3 shows a network function implementing Pulsar's rate control. We describe the state it maintains, the computation it performs and the application semantics it requires.

State. The function accesses global state—it reads from the `queueMap` to determine the rate limited queue a packet should go to.

Computation. The function ensures that the packet is rate limited based on the type of message (*i.e.*, IO operation) that it belongs to. It also queues the packet at the rate limiter corresponding to the tenant generating the packet. This ensures aggregate tenant-level guarantees instead of per-VM guarantees.

Application semantics. For a packet, the function needs to determine its message, the message size, and the tenant it belongs to.

We note that the same features apply to other schemes for datacenter QoS that offer performance guarantees to tenants and applications [9, 39, 10, 33, 52, 61].

2.1.3 Datacenter flow scheduling

Many recent proposals aim to optimize application performance in terms of reduced user-perceived latency by carefully scheduling flows across the network. For example, PIAS [8] minimizes flow completion time without application support by mimicking the shortest flow first scheduling strategy. With PIAS, flows start in the highest priority but are demoted to lower priorities as their size increases.

While PIAS [8] targets flow-level prioritization, the concept can be generalized and applied at a message level. A message could be a flow, a GET/PUT, etc. Figure 4 shows a network function implementing message-level PIAS scheduling.

State. The function uses two pieces of global state. First, `priorityThresholds` provide flow-size thresholds that define the priority for a flow given its current size. These thresholds need to be calculated periodically based on the datacenter's overall traffic load. Second, the function maintains a `msgTable` to track the current size of all active messages.

Computation. When a packet for a new message arrives, the function initializes a `messageTable` entry for it. When a packet for an existing message arrives, the

Function	Example	Data-plane state	Data-plane computation	Application semantics	Network support	Eden (out of the box)
Load Balancing	WCMP [65]	✓	✓		✗	✓
	Message-based WCMP	✓	✓	✓		
	Ananta [47]	✓	✓			
	Conga [1]	✓	✓	✓*	✓	✗
	Duet [26]	✓	✓			
Replica Selection	mcrouter [40]	✓	✓	✓	✗	✓
	SINBAD [17]	✓	✓	✓		
Datacenter QoS	Pulsar [6]	✓	✓	✓	✗	✓
	Storage QoS [61, 58]	✓	✓	✓		
	Network QoS [9, 51, 38, 33]	✓	✓	✓		
Flow scheduling and congestion control	PIAS [8]	✓	✓		✗	✓
	QJump [28]	✓	✓			
	Centralized [48, 27] congestion control	✓	✓	✓*		
	Explicit rate control (D^3 [64], PASE [45], PDQ [30])	✓	✓	✓	✓	✗
Stateful firewall	IDS(*e.g.* [19])	✓	✓		✗	✗
	Port knocking [13]	✓	✓		✗	✓

Table 1: Example network functions, their data-plane requirements and whether they need network support beyond commodity features like network priorities. Eden can support many of these functions out of the box. (✓* refers to functions that would benefit from application semantics. For example, Conga uses flowlets that approximate application messages.)

message's current size is updated. Finally, we use the current message size to determine the packet's priority.

Application semantics. PIAS achieves flow scheduling without application support. However, in closed-environments like datacenters, it is possible to modify applications or the OS to directly provide information about the size of a flow [64, 22, 30]. This would allow for shortest flow first scheduling. In Section 5, we show that this approach, by using application information, can provide increased accuracy compared to PIAS' application-agnostic scheduling.

We note that the same features apply to other schemes for optimizing application performance by reducing flow completion times [30, 45], meeting flow deadlines [64] or similar task-level metrics [18, 22].

2.2 End hosts: A natural enforcement point

Table 1 extends the previous discussion to other categories of common network functions like congestion control and stateful firewalling. The table is certainly not exhaustive; instead, it illustrates how data-plane state, computation and application semantics are common requirements across a large selection of network functions.

Given these requirements, a natural question is where should such functionality be implemented. Traditionally, network functions have been implemented at switches and middleboxes; this however reflects legacy originating from the Internet, where ISPs only control their equipment and end hosts are not trusted. By contrast, in closed settings owned and operated by a single entity, like datacenters and enterprises, end hosts (or some parts of the end hosts) are trusted. Thus, end hosts are well-suited to implement network functions—they have high computational capabilities, significant amounts of memory, can provide fine-grained visibility into application semantics, and by definition, are on the data path.

Placing network functionality at end hosts provides also additional benefits. First, enforcing a function at the source of traffic instead of distributed enforcement along its path makes it easier to achieve consistent behavior even when the function is updated. On the contrary, achieving consistent updates in the network requires complex mechanisms to ensure that all switches along the path make consistent decisions [55]. Second, implementing network functions at end hosts leads to a natural partitioning of both the state and computation overhead of enforcement because hosts are only responsible for a subset of the traffic a switch would need to process. Finally, certain network properties such as flow RTT or loss and network events like flow termination are readily available at the hosts; instead, such properties need to be inferred, typically through heuristics inside the network.

Of course, implementing network functionality at end hosts is not a panacea. As shown in Table 1, some functions do require network support beyond what is offered by commodity equipment. For example, deadline-based congestion control protocols like D^3 [64] and PDQ [30] rely on explicit, fine-grained feedback from the network. In general, network switches have better visibility of instantaneous aggregate network behavior and are needed to implement network functions that rely on such information. Furthermore, despite the resources available at end hosts, some network functions, such as compression or encryption, may be too resource intensive for general purpose processors and may require custom hardware.

In summary, a variety of network functions can indeed be implemented with a modicum of computation and

state at the end host data plane. This paper proposes an architecture to enable such functions.

3 Design

This section provides an overview of the Eden architecture. Eden targets environments in which end hosts are owned by a single administrative domain and can therefore be trusted. In this paper, we focus on datacenters, although our approach can also be applied to other scenarios such as enterprises [21, 34].

3.1 Design overview

Eden comprises three components (Figure 5):

1). *Controller.* It provides a logically centralized coordination point where any part of the network function logic requiring global visibility resides (§3.2).

2). *Stage.* Applications are a first-order citizen in Eden. Any application, library or even kernel module in the end-host stack that is Eden-compliant is called a "stage" (§3.3). Stages can *classify* packets based on stage-specific semantics, and this classification is carried through the end host stack to the Eden enclave where it is used to determine the enclave rule(s) to apply. Stages are programmed using the classification API which allows the controller to learn their classification abilities and to configure them with classification rules.

3). *Enclave.* Eden provides a programmable data-plane through an enclave at each end host (§3.4). The enclave may be implemented at the software network stack inside the OS or the hypervisor, or using programmable hardware like FPGAs and programmable NICs. The enclave is programmed by the controller using the enclave API. An enclave program comprises a set of "match-action" rules. However, instead of matching on packet headers, packets are matched based on the *class* associated with each packet. Finally, instead of a pre-defined set of actions, the action part of the match-action rule is an *action function* comprising logic that can access and modify enclave and class state.

3.2 Eden controller

A network function is conceptually a combination of a *control-plane function* residing at the controller and a *data-plane function*. Given a network function, all computation that either needs global network visibility or needs to be invoked at coarse timescales can reside in the control function. By contrast, any computation that needs to be invoked on a per-packet basis or needs to access rapidly changing state should be part of the data function. For our WCMP example (§2.1.1), the control plane function involves (periodically) determining the weight associated with each network link based on the current topology and traffic matrix. These weights are used to compute the `pathMatrix` variable used in the `WCMP` data plane function shown in Figure 2.

Recent work in the SDN literature mostly focuses on controller-based algorithms [24] and languages for expressing control plane functions [25, 62]. Hence, in the rest of this paper, we focus on how data plane functions can be implemented at stages and enclaves, and the APIs that the Eden controller uses to program such functionality.

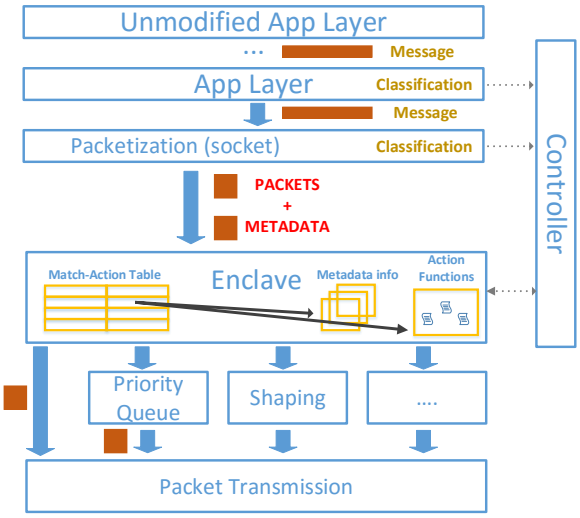

Figure 5: The Eden architecture.

3.3 Eden stages

As shown in Table 1, many network functions require application semantics as they operate at the granularity of application messages. By contrast, the Eden enclave operates at the granularity of packets. Traditional approaches infer the application message a given packet belongs to using deep packet inspection or through other heuristics. Eden adopts a different tack by allowing applications to explicitly share this information.

Applications, libraries (*e.g.*, an HTTP library) and services (*e.g.*, NFS or SMB service) that run either in user space or at the OS can opt to *classify* network traffic generated by them, i.e., identify messages and their class(es). *A* **message** *refers to an (arbitrary) application data unit while a* **class** *refers to the set of packets to which the same action function should be applied.* For example, a memcached-specific load balancing function may pick the destination server based on whether a packet corresponds to a GET or a PUT message. To allow such functions to be implemented at the Eden enclave, the memcached application needs to mark all its traffic as belonging to one of two classes, GETs and PUTs, and to identify each GET/PUT message separately. The class information is carried through the end-host stack to the enclave and is used to determine the action function to apply.

A stage specifies the application-specific fields that can be used for classification, i.e., fields that can categorize messages into classes. It also specifies *meta-data*, i.e., other application-specific data, it can generate. To make the discussion concrete, Table 2 gives examples of the classification and meta-data capabilities of three stages. For instance, an Eden-compliant memcached

Stage	Classifiers	Meta-data
memcache	<msg_type, key>	{msg_id, msg_type, key, msg_size}
HTTP library	<msg_type, url>	{msg_id, msg_type, url, msg_size}
Eden enclave	<src_ip,src_port, dst_ip, dst_port, proto >	{msg_id}

Table 2: Classification capabilities of a few stages.

$r1$: <GET, - >	→ [GET, {msg_id, msg_size}]
$r1$: <PUT, - >	→ [PUT, {msg_id, msg_size}]
$r2$: <*, - >	→ [DEFAULT, {msg_id, msg_size}]
$r3$: <GET, "a" >	→ [GETA, {msg_id, msg_size}]
$r3$: <*, "a" >	→ [A, {msg_id, msg_size}]
$r3$: <*, * >	→ [OTHER, {msg_id, msg_size}]

Figure 6: Examples of classification rules and rule-sets

client can classify its messages based on whether it is a GET or a PUT (`msg_type`) and the `key` being accessed by the message. Furthermore, the stage can associate the message with meta-data comprising a unique (message) identifier, the message type, the key being accessed, and the message size.

Beyond applications, the Eden enclave itself can also classify traffic. It operates at the granularity of packets, and like Open vSwitch, it can classify packets based on different network headers, including the IP five-tuple. This is shown in the last line of Table 2. Thus, when classification is done at the granularity of TCP flows, each transport connection is a message.

Classification rules. A stage maintains *classification rules*, written as <`classifier`> → [`class_name`, {`meta-data`}]. The `classifier` is an expression that defines the class, while the `meta-data` specifies information, including a message identifier, that should be associated with messages belonging to this class. The message identifier allows the enclave to later recover packets carrying data for the same message. Classification rules are arranged into *rule-sets* such that a message matches at most one rule in each rule-set. Rule-sets are needed since different network functions may require stages to classify their data differently. A given message can be marked as belonging to many classes, one for each rule-set.

Figure 6 lists a few example classification rules for the memcached stage. Rule-set `r1` comprises two rules, the first one matches all GET messages while the second matches PUT messages. The messages are marked as belonging to classes named "GET" and "PUT", and are associated with meta-data including a unique message identifier and the message size. Rule-set `r2` simply puts all messages into the "DEFAULT" class and associates them with a message identifier. Finally, rule-set `r3` classifies any GET requests for the key named "a" into the "GETA" class, any other request for key "a" into the "A" class, and all other requests to the "OTHER" class.

External to a stage, a class is referred to using its fully qualified name: `stage.rule-set.class_name`. Thus, the class named "GET" in rule-set `r1` is referred to as `memcached.r1.GET`. Given the rule-sets above, a PUT request for key "a" would be classified as belonging to three classes, `memcached.r1.PUT`, `memcached.r2.DEFAULT`, and `memcached.r3.A`.

S0	`getStageInfo` () returns the fields that the stage can use to classify data, and the meta-data fields it can output
S1	`createStageRule` (Rule_set s, Classifier c, Class_id i, Meta-data set m) creates classification rule <c> → [i, {m}] in rule_set s returns a unique identifer for the rule
S2	`removeStageRule` (Rule_set s, Rule_id r)

Table 3: Stage API, used by the controller to program data-plane stages.

Stage-Controller interface. The controller can program stages using the stage API shown in Table 3. Through `getStageInfo`, the controller can discover a stage's classification abilities, what fields can be used as classifiers and the meta-data it can generate. The controller can also create and remove classification rules at stages (S1,S2 calls in Table 3).

3.4 Eden enclave

The enclave resides along the end host network stack, and operates on packets being sent and received. The enclave comprises two components:

1). A set of tables with match-action rules that, depending on a packet's class, determine the action function to apply.

2). A runtime that can execute the action functions.

3.4.1 Match-action tables

In allowing data-plane programmability through enclaves, we wanted to achieve flexibility without sacrificing forwarding performance. We chose a match-action model for the enclave for two reasons. First, it is possible to efficiently implement lookup tables using TCAMs when the enclave is implemented in hardware. Secondly, thanks to the popularity of the OpenFlow community, match-action programming is already a familiar model for programmers to express network functions.

As shown in Table 4, each rule matches on class names and produces an action function which is written in a domain specific language. This is in contrast to the predefined set of actions with today's SDN design. Match-

Match	—>	Action
<match on class name>	—>	f (pkt, ...)

Table 4: Match-action table in the enclave

ing on class names allows for two types of functions. First, packet header based classification at the enclave enables functions that simply need to operate at, say, the granularity of a TCP flow. Second, classification by stages means that the enclave can implement functions that operate on a higher-level grouping of packets as defined by applications. Next, we describe the action function language and how enclaves execute the functions.

3.4.2 *Action functions*

Action functions are written in a domain specific language (DSL) built using F# code quotations.[1] The use of a DSL makes it easier to check safety properties of the function. For example, action functions cannot perform pointer arithmetic and they need to rely on limited stack and heap space. We discuss the choice of the specific language in Section 6. As summarized below, the language provides features that allow expressing basic computations including loops, and to manipulate both packets and any state at the enclave.

The action function language is a subset of F#. This subset does not include objects, exceptions, and floating point operations. Such features benefit large code bases; instead, we expect small functions running in the enclave, and hence, the overhead of implementing those features offers little value. Floating point operations are typically not supported in the kernel, and they are also not supported by the programmable NICs we experimented with. The subset of F# that we support includes basic arithmetic operations, assignments, function definitions, and basic control operations.

Action Function example. We use an example to illustrate Eden's action functions. Figure 7 shows how the PIAS function discussed in Section 2 (pseudo code in Figure 4) is written in our action function language. The function takes three parameters, `packet`, `msg` and `_global`. `packet` refers to the actual packet.

It is important to understand message in the context of the program. For an action function, a message refers to packets that should be treated as a unit by it. A message could be a TCP flow, a GET/PUT request, or a HTTP page. As mentioned above, the enclave runtime determines a packet's message identifier. The `msg` parameter thus lets the function access state shared for the processing of all packets of the same message (Section 3.4.4 details how this state is maintained). Finally, the `_global` parameter gives access to state shared by all instances of the action function that may run in parallel.

[1]https://msdn.microsoft.com/en-us/library/dd233212.aspx.

```
 1 fun(packet : Packet, msg : Message, _global :
Global) ->
 2     let msg_size = msg.Size + packet.Size
 3     msg.Size <- msg_size
 4
 5     let priorities = _global.Priorities
 6     let rec search index =
 7         if index >= priorities.Length then 0
 8         elif msg_size <=
 9             priorities.[index].MessageSizeLimit
then
10                 priorities.[index].Priority
11         else search (index + 1)
12
13     packet.Priority <-
14         let desired = msg.Priority
15         if desired < 1 then desired
16         else search 0
```

Figure 7: Program for priority selection

For each packet, the program updates the size of the flow (lines 2–3) and then searches in the `_global.Priorities` array to find the priority corresponding to the flow size (using `search`, lines 6–11). Background flows can specify a low priority class (using `flow.Priority`), and the program will respect that (lines 14–15).

An action function can have the following side-effects: (i) It can modify the `packet` variable, thus allowing the function to change header fields, (ii) It can modify the `msg` variable, including the class name and metadata associated with the message. This can be used to control routing decisions for the packet, including dropping it, sending it to a specific queue associated with rate limits, sending it to a specific match-action table or forwarding it to the controller.

We note that while the programmer accesses packet headers, as well as message and global state using the `packet`, `msg`, and `_global` function arguments respectively, this is just a convenient illusion for the programmer. It also helps during development and debugging. As we describe later, Eden's compiler rewrites the use of these function arguments.

3.4.3 *Interpreter-based execution*

A key challenge posed by executing action functions on the data plane is ensuring they can run across platforms and they can be updated dynamically by the controller without affecting forwarding performance. The latter requirement is particularly problematic, especially for today's programmable NIC platforms that do not support dynamic injection of code. Eden sidesteps these issues by using an interpreter to execute action functions. The interpreter uses a stack and heap for keeping action function state.

Additionally, the use of interpreted execution enables monitoring the execution of the action function and guaranteeing that it can read and modify only the state related to that program; *i.e.*, state not associated with

the action function is inaccessible to the program. In particular, a faulty action function will result in terminating the execution of that program, but will not affect the rest of the system. Of course, we do rely on correct execution of the interpreter, but we believe that it is easier to guarantee the correct execution of the interpreter than to verify every possible action function.

The use of the interpreter, instead of compiling to native code, hurts performance when executing the action functions. However, as we demonstrate in Section 5, this is a small penalty for the convenience of injecting code at runtime, and for the ability to use the same bytecode across platforms.

3.4.4 *Eden compiler*

We compile action functions to bytecode which is then interpreted inside the enclave. Given an action function, the most challenging aspect of the compilation process is determining its input dependencies and reflecting any changes made by the function to packet fields and the state maintained by the runtime. For instance, for the PIAS action function in Figure 7, the compiler needs to recover the function's input parameters. This includes packet information like its size, message-specific state (`msg` parameter which is used to determine message size) and global state (`_global` parameter used to determine the priority thresholds). For the output, it needs to determine how to update the packet headers (*e.g.*, 802.1q priority field in the header) or other state (*e.g.*, message size).

This is done using three kinds of *type annotations* for state variables; we expect the programmer to specify these. First, the lifetime of each state variable, *i.e.*, whether it exists for the duration of the message or till the function is being used in the enclave. Figure 8 highlights an example of the lifetime annotations (we omit the rest of the annotations due to space constraints). This lifetime information allows us to decide where to store the state requested. For convenience of state management, the enclave runtime provides the ability to store message-specific state, and makes it available upon function invocation.

Second, the access permissions of a state variable, *i.e.*, whether the function can update its value or whether it is read-only. The access permissions allows us to determine the concurrency level for the function; in Figure 7 the function can update the message size and, hence, we will process at most one packet per message concurrently. This restriction does not apply to the global state which is read-only for the function.

Finally, the mapping to packet header values; for example, the mapping of the packet size field in IPv4 headers to the "size" variable inside the function in Figure 7. Currently, we provide a fixed header parsing scheme, and are investigating more extensible parsing approaches using ideas from P4 [14].

Apart from resolving input and output dependencies, the rest of the compilation process, mainly the translation of the abstract syntax tree to bytecode, is more straightforward. Value types are preferentially placed in the stack and otherwise in the local variables and the heap. More complicated types, such as arrays, are placed in the program heap (when necessary by copying the values from the flow or function state). The layout is then used to generate the bytecode of the program. In the current version, we perform a number of optimizations such as recognizing tail recursion and compiling it as a loop.

```
[<State(lifetime = Granularity.Packet)>]
type Packet () =
  class
    [<AccessControl(Entity.PacketProcessor,
            AccessLevel.ReadOnly)>]
    [<HeaderMap("IPv4", "TotalLength")>]
    [<HeaderMap("IPv6", "PayloadLength")>]
    member val Size = 0L with get, set

    [<AccessControl(Entity.PacketProcessor,
            AccessLevel.ReadWrite)>]
    [<HeaderMap("802.1q", "PriorityCodePoint")>]
    member val Priority = 0 with get, set
  end
```

Figure 8: Example of annotations used by the priority selection policy. The HeaderMap annotations specify the field in the header that corresponds to the property. While the Packet type appears to be a generic class, we expect them to just contain properties of basic types with associated attributes. We also expect the properties to provide default initializers.

Concurrency model. As discussed, the programmer specifies the lifetime and access model of the state used by the action function with type annotations (Figure 8). The authoritative state is maintained in the enclave, and the annotations determine the concurrency model for the action functions. The enclave creates a consistent copy of the state needed by the program in the heap and stack. If the action function requires read-only access to message and global state, the enclave is allowed to schedule multiple invocations of that program in parallel (*i.e.*, the program is allowed to write only to packet state). The state stored by the enclave may change during program execution; however, the program will not see the change.

If the action function requires write-access to the state maintained at the message level, then only one packet from that message can be processed in parallel. Further, if the action function requires write-access to global state, then only one parallel invocation of the action function is allowed. These restrictions are necessary because, after program termination, the enclave must update the authoritative state with the changes, and pass updated values to the next invocation of the same action function. An alternative approach that we have not ex-

plored would be to introduce synchronization primitives in the language. This however might lead to a performance penalty while accessing shared state. Overall, the programs should restrict writable state to the finest granularity possible.

3.4.5 Enclave-Controller interface

The controller can program enclaves through the enclave API. This allows the controller to create, delete and query both tables and individual match-action rules in an enclave. We omit the API details for brevity.

3.5 Network support

Eden's end-host based approach to network functions requires two features from the network: the ability to specify a packet's priority and its route. For the former, we use 802.1q tags. For the latter, recent proposals have shown how existing technologies like MPLS and VLANs can be used to achieve source routing in datacenters [44, 16]. Here, end hosts specify the path of a network packet as a label in the packet header, *e.g.*, through an MPLS label or VLAN tag, and switches perform label-based forwarding. In our experiments, we use VLAN tagging for route control.

Such source routing requires the controller to configure the label forwarding tables at switches. For MPLS, this can be achieved through a distributed control protocol like LDP [5] while for VLANs, Spain [44] describes a solution involving multiple spanning trees. Hence, label-based forwarding and the corresponding control protocol is the primary functionality Eden requires of the underlying network. Additionally, most existing switches already support statistics gathering capabilities (e.g SNMP) and priority-based queuing.

4 Implementation

We now provide more information about the implementation of the enclave and stages including current limitations. We also describe our evaluation testbed.

4.1 Interpreter and enclave

The Eden interpreter supports relatively small programs that use limited (operand) stack and heap space. The execution is stack based, similar in spirit to the Java Virtual Machine (JVM); we support many of the standard JVM op-codes. Apart from op-codes that specify basic load and store, arithmetic, branches, and conditionals, we have also implemented as op-codes a limited set of basic functions, such as picking random numbers and accessing a high-frequency clock in the system. Unlike general VMs, we do not support many features such as exceptions, objects, and just-in-time compilation. An alternative design would be to base the interpreter on a register machine (as in [41]).

Section 3.4.4 mentioned a number of facilities expected by the enclave that we have implemented: (a) managing flow and function specific state, (b) preparing program inputs and consuming program outputs, and (c) overseeing program invocation. Overall, the enclave, including the interpreter, is about 16K lines of C code.

4.2 Stages

Stages associate class and metadata information with (application) messages; packets generated as a result of those messages will be carrying the extra information and processed by the interpreter accordingly. The implementation of that mechanism in its pure form requires a large number of changes in the current network stack, and application and library code. As a proof of concept, we have extended the socket interface to implement an additional send primitive that accepts class and metadata information (our extension works for sockets using TCP). At the kernel, we record the sequence number of the sender along with the extra information.

At the bottom of the network stack, we intercept packets before they get transmitted. When we detect that they contain data with the appropriate sequence numbers, we update the relevant message state, and schedule the packet for processing inside the enclave.

The mechanism above allows us to associate class and metadata information at a per-packet granularity. We believe that a generic mechanism for associating metadata with events that traverse application and kernel network stacks could be of general use (see also [11]).

4.3 Testbed

Our Eden implementation targets traditional OS network stacks and programmable NICs, and we thus deploy two separate testbeds.

Our *software* testbed comprises five machines, four Intel XEON W3530 (2.8GHz/12GB RAM), and one Intel XEON E7450 (2.4GHz/32GB). All machines use 10GbE Mellanox network cards, and are connected through an Arista 7050QX-32S-R switch. The enclave runs inside a Windows network filter driver. We added a new ioctl that allows the application to communicate metadata to the driver, and used that to implement the (socket-like) message send call.

Our *programmable NIC* testbed consists of four Intel Xeon E5-1620 (3.70GHz/16GB). Each machine is equipped with a dual-port 10GbE Netronome NFE-3240 programmable NIC [46]. These NICs descend from Intel's IXP network processors [31]. The NICs include a 40 core, 8 way multi-threaded CPU (320 threads), and 4GB of memory. We perform CAM operations on main memory (other models of the processor can offload them to a large TCAM). The network switch is a 40x10Gbps Blade RackSwitch.

We have extended the NIC firmware to be able to run the Eden interpreter. Beyond that, we can also execute control programs and keep state in the NIC. Currently, we have limited support for stages in our programmable NIC testbed.

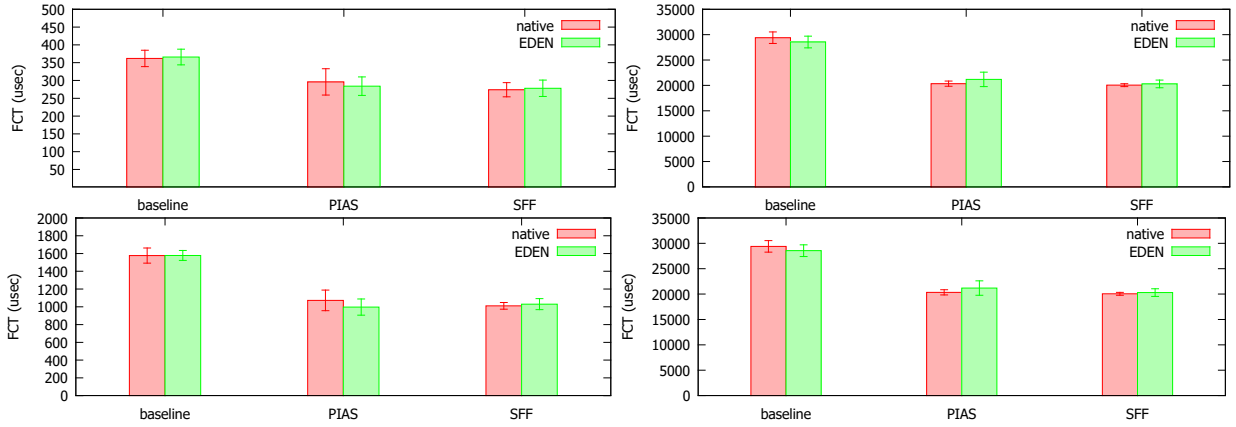

Figure 9: Average flow completion times (FCT) for small (left) and intermediate (right) flows with 95% confidence intervals. The top bars show the average while the bottom the 95^{th} percentile.

5 Evaluation

Our evaluation of Eden spans three axes: i) It highlights Eden's performance while implementing diverse network functions, ii) it covers functions implemented in an OS network stack and in programmable NICs, and iii) it provides micro-benchmarks evaluating Eden's data-plane overheads. We consider the three case studies from Section 2.

5.1 Case Study 1: Flow Scheduling

Our first case study describes how Eden implements network functions that schedule flows so as to reduce flow completion times. We examine two such functions: PIAS, whose logic was discussed in Figure 4 and shortest flow first scheduling (SFF). PIAS requires data-plane computation and state to enable tracking flow sizes and tagging packets with priorities depending on thresholds based on the distribution of flow sizes. SFF, instead, does not track traffic flow sizes but it requires applications to provide the flow size to the Eden enclave, so that the correct priority is enforced.

To assess the impact of Eden on the performance of these functions, we compare Eden with a "native" implementation. The latter implements a hard-coded function within the Eden enclave instead of using the interpreter, similar to a typical implementation through a customised layer in the OS [8]. Figure 7 shows the PIAS action function in our language.

The workload driving the experiments is based on a realistic request-response workload, with responses reflecting the flow size distribution found in search applications [2, 8]. Such applications generate traffic mostly comprising small flows of a few packets with high rate of flows starting and terminating. In our set-up, one worker responds to requests generating load at roughly 70%, while other sources generate background traffic at the same time. Priority thresholds were set up for three classes of flows: small (<10KB), intermediate (10KB-1MB) and background. Small flows have the highest priority, followed by the intermediate ones.

Figure 9 shows the average and the 95^{th}-percentile of the flow completion times for small and intermediate flows when executing the two functions. The figure shows results when running natively in the OS and through the Eden interpreter across ten runs of the experiments. *Baseline* highlights flow completion times without any prioritization. We report two Baseline figures, native and Eden. The latter shows the overhead of running the classification and data-plane functions, but ignoring the interpreter output before packets are transmitted (i.e., the priority marks are not inserted in the packet header). Essentially, the overhead reflects running the Eden enclave over vanilla TCP.

As expected, enabling prioritization significantly reduces the flow completion times; for small flows, flow completion times reduce from $363\mu s$ to $274\mu s$ on the average, and from $1.6ms$ to $1ms$ at the 95^{th} percentile, an overall a reduction of 25%-40%. Similar trends are observed for intermediate flows as well. At the same time, background traffic manages to saturate the rest of the link capacity. SFF, by utilizing application knowledge, typically provides slightly better performance with less variability. In SFF, the mapping of flows to classes occurs when the flow starts, and flows do not change priorities over time.

While similar observations have been reported in previous work, what is important in our context is that the performance of the native implementation of the policy and the interpreted one are similar. In all cases the differences are not statistically significant.

5.2 Case Study 2: Load-balancing

The second case study examines how Eden enables per-packet WCMP on top of our *programmable NIC* testbed. The WCMP function has been discussed in Section 2 (Figure 2). We arranged the topology of our testbed to emulate the topology of Figure 1, with two hosts connected through two paths, one 10Gbps and one 1Gbps.

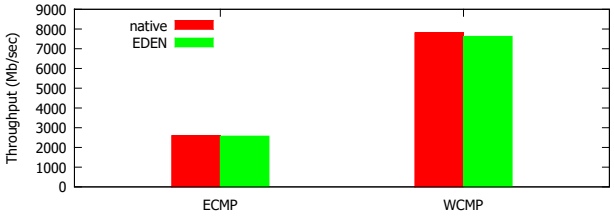

Figure 10: Aggregate throughput for ECMP and WCMP. Confidence intervals are within 2% of the values shown.

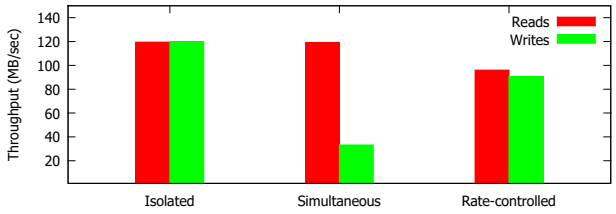

Figure 11: Average READ vs. WRITE throughput when requests run in isolation, simultaneously, and when READ requests are rate-controlled based on the request size. 95% confidence intervals are <0.2% of the mean values.

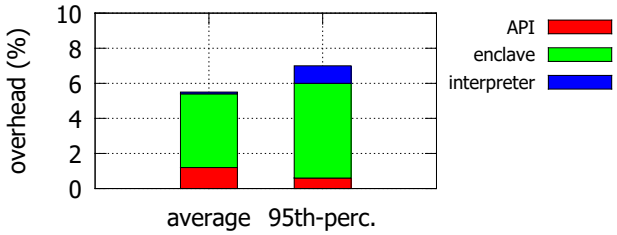

Figure 12: CPU overheads incurred by Eden components compared to running the vanilla TCP stack. API refers to passing metadata information to the enclave.

The programmable NICs run our custom firmware that implements the enclave and the Eden interpreter. The interpreted program controls how packets are source-routed through the two paths. We use per-packet balancing, instead of per-flow balancing to highlight Eden's ability to apply per-packet functions at high rate. In the default case, paths are selected for long-running TCP flows with equal weights, thus implementing ECMP. For WCMP, we enforce a ratio 10:1.

Figure 10 shows the average throughput achieved for the two functions with a native program implemented directly on the programmable NICs and with Eden. For both ECMP and WCMP, Eden's overheads are negligible with the difference between Eden and native not being statistically significant. In ECMP, TCP throughput is dominated by the capacity of the slowest path, and throughput peaks at just over 2Gbps as expected. Instead, with per-packet WCMP, TCP throughput is around 7.8Gbps, 3x better than ECMP. The throughput is lower than the full 11Gbps which is our topology's min cut due to in-network reordering of packets [29]. Modifying TCP's congestion control algorithm can alleviate such issues [53]; however, the goal of this case-study is to highlight how Eden can easily implement network functions such as weighted selection of paths without any impact on performance–in this case with unmodified applications and while running vanilla TCP.

5.3 Case Study 3: Datacenter QoS

We now examine Pulsar's rate control as described in Figure 3. The experiment involves two tenants running our custom application that generates 64K IOs. One of the tenants generates READ requests while the other one WRITEs to a storage server backed by a RAM disk drive. The storage server is connected to our testbed through a 1Gbps link.

The results in Figure 11 show that when WRITE requests compete with READs, their throughput drops by 72%. As previously discussed, this reflects the asymmetry of IO operations; READs are small on the forward path and manage to fill the queues in shared resources. Instead, we account for this mismatch through Pulsar's rate control, by charging READ requests based on the request size and WRITEs on the packet size. This ensures equal throughput between the two operations.

5.4 Overheads

The previous case studies have focused on the overheads incurred by Eden with respect to application metrics. Here, we quantify Eden's CPU overheads and memory footprint of the enclave.

Figure 12 quantifies Eden overheads while running the SFF policy (case study 1). The workload comprises 12 long-running TCP flows in one of our testbed nodes, and Eden can saturate the 10Gbps capacity of its link. The figure shows how the different components of Eden contribute to the CPU load. We consider these overheads reasonable. Note however, that saturating the link capacity assumes that the cycle budget of the interpreted program allows for such speeds. As we discuss in the following section, Eden intentionally avoids posing any restrictions in the cycle budget of data-plane functions.

In the examples discussed in the paper, the (operand) stack and heap space of the interpreter are in the order of 64 and 256 bytes respectively.

6 Discussion

The previous sections discussed Eden's design and demonstrated its expressiveness through case studies. Here, we briefly complement the discussion of our design choices and highlight interesting issues for future research.

Choice of language and environment. Beyond the benefits discussed in Section 3.4 (*i.e.*, expressiveness, portability, safety, dynamically-injected programs), the choice of the DSL and the interpreted environment were

further motivated by fast prototyping. In particular, the use of F# code quotations is convenient because it facilitates retrieval of the abstract syntax tree of the program, which is the input to our compiler. Additionally, F# allowed us to experiment with various ways of expressing the action functions, and to run and debug the programs locally (that can even take place with the F# interpreter without invoking the compiler and the enclave interpreter).

Action function composition. In this paper, we assumed a fixed order of network function execution. This is effective when the functions run in isolation, or when their composition can be determined trivially. Network functions, however, can interact in arbitrary ways, hence, it is an open question to define the semantics of function composition. One option is to impose a hierarchy. This can be achieved, for example, by imposing that application specific functions precede flow specific ones or vice versa, or apply priorities to functions which define the execution order.

OS vs NIC program execution. Eden's interpreter can run both in the kernel and in a programmable NIC. When both are available, deciding where functions run is an open question. While it may seem that processing should always be offloaded, the cost of offloading may be prohibitive in some cases. For example, when the associated metadata is large, or when the interaction with the local controller is frequent. Automatically partitioning of a high-level policy across different execution points would thus be very useful.

Cycle budget and memory requirements. Achieving line rates of 10Gbps+ in modern datacenter networks results in a tight cycle budget for data plane computations. Techniques like IO batching and offloading are often employed to reduce the processing overhead. While Eden's action functions express per-packet computation, they can be extended to allow for computation over a batch of packets. If the batch contains packets from multiple messages, the enclave will have to preprocess it and split it into messages.

While the enclave can, in principle, limit the amount or resources (memory and computational cycles) used by an action function, we chose not to restrict the complexity of the computation or the amount of state maintained by them. Instead, we believe that the system administrator should decide whether to run a function or not, irrespective of its overheads. Indeed, deeming such overheads acceptable may depend on exogenous factors such as the expected load in the system.

7 Related work

The idea of a programmable data plane dates back to active networking. Active networks exposed an API that allows control over router and switch resources, enabling processing, storage and packet queue management [60, 56, 63, 12, 3]. However, it has proven difficult to realize a generic programming model implemented by networking devices. Eden avoids this issue by focusing on data plane functions executed by end hosts only. Recently, Jeyakumar et al. proposed the use of tiny programs that can be carried in packets and executed by routers [32]. This provides a flexible mechanism to exchange state information between the network and the end hosts. Eden can benefit from such information, but is agnostic to how it is obtained.

Today, programmable networks are implemented using Software-Defined Networking (SDN) [23, 37]. SDN decouples the control plane from the data plane and requires a way for the control plane to communicate with the data plane programmatically. The most prominent API for such control- and data-plane interaction is OpenFlow [42]. Eden focuses on implementing functionality at the ends with a wider and richer data-plane interface, and goes one step further to take advantage of application knowledge.

P4 [14] proposes a language for programming packet processors, hence, generalizing OpenFlow [42]. P4 targets modern switching chips, *e.g.*, [15], and is tailored to a specific abstract forwarding model comprising parse, match and action. This results in a restricted programming model that limits its expressivity; for example, to policies that can be encoded through exact matches. Using P4 to encode a policy involving comparison operations, like the flow scheduling example used throughout this paper, would be challenging.

The idea of end hosts participating in network functionality has been explored in enterprise and data center networks [49, 16, 21, 34, 59, 57]. For example, Open vSwitch [50] extends the OpenFlow [42] programming model to end hosts. However, even in these proposals, the data-plane interface is narrow—it can be programmed to match (mostly) on packet headers and apply a fixed set of actions. They can thus implement application-agnostic network functions that operate at coarse time-scales. Instead, Eden can accommodate network functions that are dynamic, require application semantics, and operate even at per-packet time-scales.

Network exception handlers [34] was one of the first proposals to incorporate application semantics into management policies. Akin to program exception handlers, end hosts register exception conditions with network controllers, which in turn inform end hosts when exceptions trigger. Corresponding actions at end hosts would then be parameterized with application context. Applications are a first-order citizen in Eden as well; however, Eden goes beyond static actions such as rate limiting, by allowing for functions to be evaluated at end hosts with minimal controller involvement.

Network function virtualization (NFV) targets virtualizing arbitrary network functions, traditionally implemented at specialized physical devices. Eden can be seen as an end host pragmatic and reasoned point in the NFV space.

The Click software router [36] allows the construction of modular data-plane processing pipelines. It empha-

sizes the flexible chaining of building blocks called elements. Eden also operates in the data-plane, but leverages application information to enables application-level message-oriented policies. The enclaves share similarities with Click's elements. Eden's enclave offers the convenience of a higher-level environment to network policy developers. The elements, however, are more appropriate for performance critical tasks (*e.g.*, rate limiting functionality).

Eden enables rapid prototyping of network functions in the data plane. Other efforts to enable innovation in the data plane include PLUG [20] and SwitchBlade [7]. These efforts focus on implementing full protocols like IPv6 on programmable hardware like FPGAs.

Similarly to packet filters [43, 41], the Eden interpreter processes packets in a constrained execution environment inside the kernel according to user-defined programs. However, the Eden interpreter supports richer packet processing to accommodate more expressive action functions.

The idea of associating metadata with network traffic as it gets processed in the network stack is similar to Mbuf_tags [35]. This work also proposes a user-level mechanism to attach metadata to packets. Eden aims to start the discussion of extending the interface to higher level libraries and applications.

8 Concluding remarks

This paper proposes Eden, an architecture for implementing network functions at end hosts. This involves applications (stages) classifying application data into messages and annotating the messages with meta-data. Enclaves use the message and meta-data information to apply generic stateful actions on network packets.

Network operators express desired actions in a high-level language based on F# code quotations; their programs are compiled to bytecode suitable for execution by an interpreter that operates in the enclave. The enclave compiler and runtime takes care of managing any state in the enclave, freeing the network programmer to focus on the functionality to be achieved. The combination of stages, enclave and the interpreted action language allows expressing a variety of functions in a natural way and with a small performance penalty.

Acknowledgments

We are indebted to our colleague Andy Slowey for providing technical support with our experimental setup. We are also grateful to Rolf Neugebauer, Stuart Wray, and their colleagues at Netronome for their help with the NFP3240 NICs. Finally, we thank Dimitrios Vytiniotis, Z. Morley Mao and the anonymous SIGCOMM reviewers for their feedback.

References

[1] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In: SIGCOMM. ACM, 2014.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan Data Center TCP (DCTCP). In: SIGCOMM. ACM, 2010.

[3] K. G. Anagnostakis, M. W. Hicks, S. Ioannidis, A. D. Keromytis, and J. M. Smith Scalable Resource Control in Active Networks. In: IWAN. Springer-Verlag, 2000.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris Reining in the Outliers in Map-reduce Clusters Using Mantri. In: OSDI. USENIX, 2010.

[5] L. Andersson, P. Doolan, N. Feldman, A. Fredette, and B. Thomas *LDP Specification*. RFC 3036. 2001.

[6] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska End-to-end Performance Isolation Through Virtual Datacenters. In: OSDI. USENIX, 2014.

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

[8] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian Information-Agnostic Flow Scheduling for Commodity Data Centers. In: NSDI. USENIX, 2015.

[9] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron Towards Predictable Datacenter Networks. In: SIGCOMM. ACM, 2011.

[10] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea Chatty Tenants and the Cloud Network Sharing Problem. In: NSDI. USENIX, 2013.

[11] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier Using Magpie for Request Extraction and Workload Modelling. In: OSDI. USENIX, 2004.

[12] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura An Architecture for Active Networking. In: HPN. Chapman & Hall, Ltd., 1997.

[13] G. Bianchi, M. Bonola, A. Capone, and C. Cascone OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *SIGCOMM Comput. Commun. Rev.* 44, 2 (2014).

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (2014).

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

[16] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian Fabric: A Retrospective on Evolving SDN. In: HotSDN. ACM, 2012.

[17] M. Chowdhury, S. Kandula, and I. Stoica Leveraging Endpoint Flexibility in Data-intensive Clusters. In: SIGCOMM. ACM, 2013.

[18] M. Chowdhury, Y. Zhong, and I. Stoica Efficient Coflow Scheduling with Varys. In: SIGCOMM. ACM, 2014.

[19] Cisco *Snort*. 2015. url: https://www.snort.org/ (visited on 06/03/2015).

[20] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers. In: SIGCOMM. ACM, 2009.

[21] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy ETTM: A Scalable Fault Tolerant Network Manager. In: NSDI. USENIX, 2011.

[22] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron Decentralized Task-aware Scheduling for Data Center Networks. In: SIGCOMM. 2014.

[23] N. Feamster, J. Rexford, and E. Zegura The Road to SDN: An Intellectual History of Programmable Networks. *SIGCOMM Comput. Commun. Rev.* 44, 2 (2014).

[24] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi Participatory Networking: An API for Application Control of SDNs. In: SIGCOMM. ACM, 2013.

[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker Frenetic: A Network Programming Language. In: ICFP. ACM, 2011.

[26] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang Duet: Cloud Scale Load Balancing with Hardware and Software. In: SIGCOMM. ACM, 2014.

[27] C. Gkantsidis, T. Karagiannis, P. Key, B. Radunovic, E. Raftopoulos, and D. Manjunath Traffic Management and Resource Allocation in Small Wired/Wireless Networks. In: CoNEXT. ACM, 2009.

[28] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft Queues Don′t Matter When You Can JUMP Them! In: NSDI. USENIX Association, 2015.

[29] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley Multi-path TCP: A Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet. *IEEE/ACM Trans. Netw.* 14, 6 (2006).

[30] C.-Y. Hong, M. Caesar, and P. B. Godfrey Finishing Flows Quickly with Preemptive Scheduling. In: SIGCOMM. ACM, 2012.

[31] D. F. Hooper *Using IXP2400/2800 Development Tools. A Hands-on Approach to Network Processor Software Design.* 1st ed. Intel Press, 2005.

[32] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières Tiny Packet Programs for Low-latency Network Control and Monitoring. In: HotNets-XII. ACM, 2013.

[33] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg EyeQ: Practical Network Performance Isolation at the Edge. In: NSDI. USENIX, 2013.

[34] T. Karagiannis, R. Mortier, and A. Rowstron Network Exception Handlers: Host-network Control in Enterprise Networks. In: SIGCOMM. ACM, 2008.

[35] A. D. Keromytis Tagging Data in the Network Stack: Mbuf_Tags. In: BSDC'03. USENIX Association, 2003.

[36] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (2000).

[37] D. Kreutz, F. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE* 103, 1 (2015).

[38] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM Comput. Commun. Rev.* 42, 3 (2012).

[39] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma Application-driven Bandwidth Guarantees in Datacenters. In: SIGCOMM. 2014.

[40] A. Likhtarov, R. Nishtala, R. McElroy, H. Fugal, A. Grynenko, and V. Venkataramani *Introducing mcrouter: A memcached protocol router for scaling memcached deployments.* Facebook. 2014. URL: http://bit.ly/1TpNKo0 (visited on 06/16/2015).

[41] S. McCanne, and V. Jacobson The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: USENIX Winter Conf. USENIX, 1993.

[42] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (2008).

[43] J. Mogul, R. Rashid, and M. Accetta The Packer Filter: An Efficient Mechanism for User-level Network Code. In: SOSP '87. ACM, 1987.

[44] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul SPAIN: COTS Data-center Ethernet for Multipathing over Arbitrary Topologies. In: NSDI. USENIX, 2010.

[45] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar Friends, Not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In: SIGCOMM. ACM, 2014.

[46] Netronome *Netronome FlowNICs.* 2015. URL: http://netronome.com/product/flownics/ (visited on 06/03/2015).

[47] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri Ananta: Cloud Scale Load Balancing. In: SIGCOMM. ACM, 2013.

[48] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal Fastpass: A Centralized "Zero-queue" Datacenter Network. In: SIGCOMM. ACM, 2014.

[49] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker Extending Networking into the Virtualization Layer. In: HotNets-VIII. ACM, 2009.

[50] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado The Design and Implementation of Open vSwitch. In: NSDI. USENIX, 2015.

[51] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica FairCloud: Sharing the Network in Cloud Computing. In: HotNets-X. ACM, 2011.

[52] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing. In: SIGCOMM. ACM, 2013.

[53] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In: NSDI. USENIX, 2012.

[54] M. Reitblatt, M. Canini, A. Guha, and N. Foster FatTire: Declarative Fault Tolerance for Software-Defined Networks. In: HotSDN. ACM, 2013.

[55] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker Abstractions for Network Update. In: SIGCOMM. ACM, 2012.

[56] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge Smart Packets: Applying Active Networks to Network Management. *ACM Trans. Comput. Syst.* 18, 1 (2000).

[57] A. Shieh, S. Kandula, and E. G. Sirer SideCar: Building Programmable Datacenter Networks Without Programmable Switches. In: Hotnets-IX. ACM, 2010.

[58] D. Shue, M. J. Freedman, and A. Shaikh Performance Isolation and Fairness for Multi-tenant Cloud Storage. In: OSDI. USENIX, 2012.

[59] R. Soulé, S. Basu, R. Kleinberg, E. G. Sirer, and N. Foster Managing the Network with Merlin. In: HotNets-XII. ACM, 2013.

[60] D. L. Tennenhouse, and D. J. Wetherall Towards an Active Network Architecture. *SIGCOMM Comput. Commun. Rev.* 26, 2 (1996).

[61] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu IOFlow: A Software-defined Storage Architecture. In: SOSP. ACM, 2013.

[62] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak Maple: Simplifying SDN Programming Using Algorithmic Policies. *SIGCOMM Comput. Commun. Rev.* 43, 4 (2013).

[63] D. Wetherall, J. V. Guttag, and D. Tennenhouse ANTS: a toolkit for building and dynamically deploying network protocols. In: OPENARCH. IEEE, 1998.

[64] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron Better Never Than Late: Meeting Deadlines in Datacenter Networks. In: SIGCOMM. ACM, 2011.

[65] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In: EuroSys. ACM, 2014.