NetBricks: Taking the V out of NFV

Aurojit Panda and Sangjin Han, *University of California, Berkeley;* Keon Jang, *Google;*
Melvin Walls and Sylvia Ratnasamy, *University of California, Berkeley;*
Scott Shenker, *University of California, Berkeley, and International Computer Science Institute*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

NetBricks: Taking the V out of NFV

Aurojit Panda[†] Sangjin Han[†] Keon Jang[‡] Melvin Walls[†] Sylvia Ratnasamy[†] Scott Shenker[†⋆]
[†] UC Berkeley [‡] Google [⋆] ICSI

Abstract

The move from hardware middleboxes to software network functions, as advocated by NFV, has proven more challenging than expected. Developing new NFs remains a tedious process, requiring that developers repeatedly rediscover and reapply the same set of optimizations, while current techniques for providing isolation between NFs (using VMs or containers) incur high performance overheads. In this paper we describe NetBricks, a new NFV framework that tackles both these problems. For building NFs we take inspiration from modern data analytics frameworks (*e.g.,* Spark and Dryad) and build a small set of customizable network processing elements. We also embrace type checking and safe runtimes to provide isolation in software, rather than rely on hardware isolation. NetBricks provides the same memory isolation as containers and VMs, without incurring the same performance penalties. To improve I/O efficiency, we introduce a novel technique called zero-copy software isolation.

1 Introduction

Networks today are responsible for more than just forwarding packets, this additional functionality is implemented using "middleboxes". Middleboxes implement a wide range of functionality, including security (*e.g.,* firewalls, IDS/IPSs), performance (*e.g.,* caches, WAN optimizers) and support for new applications and protocols (*e.g.,* TLS proxies). Middlebox functionality was initially provided by dedicated hardware devices, and is in wide deployment today. A 2012 survey [44] found that in many networks there are equal numbers of middleboxes, switches and routers.

Approximately four years ago, many large carriers initiated an effort, called Network Function Virtualization (NFV), to replace hardware middleboxes with software implementations running in VMs [10]. This approach enabled middlebox functionality (called *Network Functions* or NFs) to be run on commodity servers and was supposed to bring several advantages including: (a) simplifying deployment, since deploying new functionality merely requires software changes; (b) simpler management using standard tools for managing VMs; (c) faster development, which now requires writing software that runs on commodity hardware; and (d) reduced costs by consolidating several NFs on a single machine. However, despite these promised advances, there has been little progress towards large-scale NF deployments. Our discussions with three major carriers revealed that they are only just beginning small scale test deployments (with 10-100s of customers) using simple NFs *e.g.,* firewalls and NATs.

The move from hardware middleboxes to software NFs was supposed to speed innovation, so why has progress been so slow? We believe this delay is because traditional approaches for both *building* and *running* NFs are a poor match for carrier networks, which have the following requirements: *performance*, NF deployments should be able to provide per-packet latencies on the order of 10s of µs, and throughput on the order of 10s of Gbps; *efficiency*, it should be possible to consolidate several NFs on a single machine; support for *chaining*, since each packet is typically processed by a sequence of NFs; the flexibility to run NFs manufactured by *multiple vendors*; and the ability to process packets from *multiple tenants* while providing some degree of isolation between them. Note that because many carriers provide middlebox services to their customers, the NFs supported by carriers include those that are commonly found in enterprise environments (*e.g.,* firewalls, NATs, IPS/IDSs, WAN optimizers, etc.) in addition to ones specific to carriers (*e.g.,* EPC, carrier-grade NAT).

Why do current tools for building and running NFs fall short of these requirements? In terms of *building* NFs, tools need to support both rapid-development (achieved through the use of high-level abstractions) and high performance (often requiring low-level optimizations). In other application domains, programming frameworks and models have been developed to allow developers to use high-level abstractions while the framework optimizes the implementations of those abstractions (ensuring high performance); the rise of data analytic frameworks (*e.g.,* Hadoop, Spark) is an example of this phenomenon. However, the state-of-the-art for NFV is much more primitive. There are programming models such as Click [27] that do not provide easily customizable low-level optimizations, and libraries such as

DPDK [23] that only provide highly-optimized packet I/O and low-level processing primitives (which alone is not sufficient to implement real-world NFs), but no approach that provides both high performance and rapid development. The result is that today NFV developers typically spend much time optimizing their code, which greatly slows development time and increases the likelihood for bugs.

The current approaches for *running* NFs is also inadequate. Isolation between NF is critical: memory isolation is essential for ensuring safety between NFs (which might come from different vendors); and performance isolation is essential to allow such deployments to serve multiple customers. Currently, NFV deployments rely on VMs and containers to provide isolation, but as we show in §5, they incur substantial performance overheads for simple NFs.

To address these inadequacies in the current NFV paradigm, we propose a very different approach for building and running NFs.[1] Our approach, called NetBricks, is clean-slate in that it requires rewriting NFs, but we do not see this as a significant drawback given the relative lack of progress towards NFV deployments. Our approach targets deployments in large carrier networks, but applies to other environments as well.

NetBricks provides both a programming model (for building NFs) and an execution environment (for running NFs). The programming model is built around a core set of high-level but customizable abstractions for common packet processing tasks; to demonstrate the generality of these abstractions and the efficiency of their implementations, we reimplemented 5 existing NFs in NetBricks and show that they perform well compared to their native versions. Our execution environment relies on the use of safe languages and runtimes for memory and fault isolation (similar to existing systems we rely on scheduling for performance isolation). Inter-process communication is also important in NF deployments, and IPC in these deployments must ensure that messages cannot be modified by an NF after being sent, a property we refer to as packet isolation. Current systems copy packets to ensure packet isolation, we instead use static check to provide this property without copies. The resulting design, which we call Zero-Copy Software Isolation (ZCSI), is the first to achieve memory and packet isolation with *no* performance penalty (in stark contrast to virtualization).

NetBricks is open source and is available at `https://netbricks.io`.

2 Background and Motivation

In this section we provide a few more details on the problems with today's approaches to NFV, and then give a high-level description of how NetBricks resolves these problems. As in much of this paper, we separate the task of building NFs from the task of running them.

2.1 Building NFs

The vast majority of commercial NFs today make use of a fast I/O library (DPDK, netmap, etc.). While this greatly improves I/O performance, developers are responsible for all other code optimizations. The Click modular router (which can also make use of such libraries) enables developers to construct an NF by connecting together various packet processing modules (called elements). While Click does not limit how packets flow between elements, modules typically support only limited amount of customization through setting various parameters. Thus, when implementing new NF functionality, developers commonly need to implement new modules, and optimizing the performance of such a module is difficult and time-consuming.

Our approach differs in two respects. First, we limit the set of such modules to core functions such as packet parsing, processing payloads, bytestream processing, and the like. That is, rather than have developers deal with a large set of modules – trying to determine which best suit their needs in terms of optimization and generality – NetBricks focuses on a core set with well-known semantics and highly-optimized implementations.

Second, in order to provide the necessary generality, we allow these core modules to be customized through the use of User-Defined Functions (UDFs). This gives these modules great flexibility, while allowing NetBricks to use optimized implementations of these modules. We think this approach represents a sweet-spot in the tradeoff between flexibility and performance; yes, one can imagine NFs that would not be easily supported by the set of modules NetBricks provides, but all the common NFs we know of fit comfortably within NetBricks' range. NetBricks thus gives developers the flexibility they need, and operators the performance they want.

One can think of the relationship between Click and NetBricks to be analogous to the difference between MPI and Map Reduce. Both Click and MPI give developers a totally general framework in which to build their applications, but the developer must take on the task of optimizing the resulting code (unless they can reuse existing modules without change). In contrast, NetBricks and Map Reduce support only a more limited set of abstractions whose actions can be customized through user code.

[1] Note that in addition to building and running NFs, one also has to *manage* them. There are separate and active efforts on this topic (discussed in §6) in both research [12, 37] and industry [11, 30] that are orthogonal to our concerns here.

2.2 Running NFs

Current NFV deployments typically involve NFs running in containers or VMs, which are then connected via vSwitch. In this setup VMs and containers provide isolation, ensuring that one NF cannot access memory belonging to another and the failure of an NF does not bring down another. The vSwitch abstracts NICs, so that multiple NFs can independently access the network, and is also responsible for transferring packets between NFs. This allows several NFs to be consolidated on a single physical machine and allows operators to "chain" several NFs together *i.e.,* ensure packets output from one NF are sent to another.

However these mechanisms carry a significant performance penalty. When compared to a single process with access to a dedicated NIC, per-core throughput drops by up to 3× when processing 64B (minimum size) packets using containers, and by up to 7× when using VMs. This gap widens when NFs are chained together; containers are up to 7× slower than a case where all NFs run in the same process, and VMs are up to 11× slower. Finally, running chaining multiple NFs in a single process is up to 6× faster than a case where each NF runs in a container (or VM) and is allocated its own core – this shows that adding cores does not address this performance gap. We provide more details on these results in §5.3.

The primary reason for this performance difference it that during network I/O packets must cross a hardware memory isolation boundary. This entails a context switch (or syscall), or requires that packets must cross core boundaries; both of which incur significant overheads. We avoid these overheads by relying on compile-time and runtime checks to enforce memory isolation in software. This is similar what was proposed by Singularity [20]. To further reduce packet I/O costs we use unique types [13] to implement safe 0-copy packet I/O between NFs. We call this technique Zero-Copy Software Isolation (ZCSI) and show that it provides low-overhead isolation for NFs.

3 Design

In this section we describe the design of NetBricks, starting with the programming abstractions and ending with the execution environment. We focus on NetBricks's architecture in this section, and present implementation notes in the next section.

3.1 Programming Abstractions

Network functions in NetBricks are built around several basic abstractions, whose behavior is dictated by user supplied functions (UDFs). An NF is specified as a directed graph with these abstractions as nodes. These abstractions fall into five basic categories – packet processing, bytestream processing, control flow, state management, and event scheduling – which we now discuss in turn.

Abstractions for Packet Processing: Each packet in NetBricks is represented by a structure containing (i) a stack of headers; (ii) the payload; and (iii) a reference to any per-packet metadata. Headers in NetBricks are structures which include a function for computing the length of the header based on its contents. Per-packet metadata is computed (and allocated) by UDFs and is used to pass information between nodes in an NF. UDFs operating on a packet are provided with the packet structure, and can access the last parsed header, along with the payload and any associated metadata. Each packet's header stack initially contains a "null" header that occupies 0 bytes.

We provide the following packet processing operators:

- **Parse:** Takes as input a header type and a packet structure (as described above). The abstraction parses the payload using the header type and pushes the resulting header onto the header stack and removes bytes representing the header from the payload.
- **Deparse:** Pops the bottom most header from the packet's header stack and returns it to the payload.
- **Transform:** This allows the header and/or payload to be modified as specified by a UDF. The UDF can make arbitrary changes to the packet header and payload, change packet size (adding or removing bytes from the payload) and can change the metadata or associate new metadata with the packet.
- **Filter:** This allows packet's meeting some criterion to be dropped. UDFs supplied to the filter abstraction return true or false. Filter nodes drops all packets for which the UDF returns false.

Abstractions for Bytestream Processing: UDFs operating on bytestreams are given a byte array and a flow structure (indicating the connection). We provide two operators for bytestream processing:

- **Window:** This abstraction takes four input parameters: window size, sliding increment, timeout and a stream UDF. The abstraction is responsible for receiving, reordering and buffering packets to reconstruct a TCP stream. The UDF is called whenever enough data has been received to form a window of the appropriate size. When a connection is closed or the supplied timeout expires, the UDF is called with all available bytes. By default, the Window abstraction also forwards all received packets (unmodified), allowing windows to be processed outside of the regular datapath. Alternatively, the operator can drop all received packets, and generate and send a modified output stream using the packetize node.
- **Packetize:** This abstraction allows users to convert

byte arrays into packets. Given a byte array and a header stack, the implementation segments the data into packets with the appropriate header(s) attached.

Our current implementations of these operators assume the use of TCP (*i.e.,* we use the TCP sequence numbers to do reordering, use FIN packets to detect a connection closing, and the *Packetize* abstraction applies headers by updating the appropriate TCP header fields), but we plan to generalize this to other protocols in the future.

Abstractions for Control Flow Control flow abstractions in NetBricks are necessary for branching (and merging branches) in the NF graph. Branching is required to implement conditionals (*e.g.,* splitting packets according to the destination port, etc.), and for scaling packet processing across cores. To efficiently scale across multiple cores, NFs need to minimize cross-core access to data (to avoid costs due to cache effects and synchronization); however, how traffic should be partitioned to meet this objective depends on the NF in question. Branching constructs in NetBricks therefore provide NF authors a mechanism to specify an appropriate partitioning scheme (*e.g.,* by port or destination address or connection or as specified by a UDF) that can be used by NetBricks's runtime. Furthermore, branching is often also necessary when chaining NFs together. Operators can use NetBricks's control flow abstractions to express such chaining behavior by dictating which NF a packet should be directed to next. To accomplish these various goals, NetBricks offers three control flow operators:

- **Group By:** Group By is used either to explicitly branch control flow within an NF or express branches in how multiple NFs are chained together. The group by abstraction takes as input the number of groups into which packets are split and a packet-based UDF which given a packet returns the ID of the group to which it belongs. NetBricks also provides a set of predefined grouping functions that group traffic using commonly-used criterion (*e.g.,* TCP flow).
- **Shuffle:** Shuffles is similar to Group By except that the number of output branches depends on the number of active cores. The runtime uses the group ID output by the shuffle node to decide the core on which the rest of the processing for the packet will be run. Similar to Group By, NF writers can use both user-defined functions and predefined functions with shuffle nodes. Semantically, the main difference lies in the fact that shuffle outputs are processed on other cores, and the number of outputs is not known at compile time.
- **Merge:** Merge provides a node where separate processing branches can be merged together. All packets entering a merge node exit as a single group.

State Abstraction Modern processors can cheaply provide consistent (serializable) access to data within a core; however, cross-core access comes at a performance cost because of the communication required for cache coherence and the inherent cost of using synchronization primitives such as locks. As a result, NFs are commonly programmed to partition state and avoid such cross-core accesses when possible, or use looser consistency (reducing the frequency of such accesses) when state is not partitionable in this way. Rather than requiring NF writers to partition state and reason about how to implement their desired consistency guarantees, NetBricks provides state abstractions.

Our state abstractions partition the data across cores. Accesses within a core are always synchronized, but we provide several options for other accesses, including (a) no-external-access, *i.e.,* only one core accesses each partition; (b) bounded inconsistency where only one core can write to a partition, but other cores can read these writes within a user supplied bound (specified as number of updates); and (c) strict-consistency where we use traditional synchronization mechanisms to support serializable multi-reader, multi-writer access.

Abstractions for Scheduled Events We also support invocation nodes, which provide a means to run arbitrary UDFs at a given time (or periodically), and can be used to perform tasks beyond packet processing (*e.g.,* collect statistics from a monitoring NF).

3.2 Execution Environment

Next we describe NetBricks's runtime environment, which is responsible for providing isolation between NFs, and NF placement and scheduling.

Isolation As we discuss in §5, container and VM based isolation comes at a significant penalty for simple NFs (for very complex NFs, the processing time inside the NF dominates all other factors, and this is where the efficiency of the NFs built with NetBricks becomes critical). NetBricks therefore takes a different tack and uses software isolation. Previously, Singularity [20] showed that the use of safe languages (*i.e.,* ones which enforce certain type checks) and runtimes can be used to provide memory isolation that is equivalent to what is provided by the hardware memory management unit (MMU) today. NetBricks borrows these ideas and builds on a safe language (Rust) and uses LLVM [28] as our runtime. Safe languages and runtime environments provide four guarantees that are crucial for providing memory isolation in software: (a) they disallow pointer arithmetic, and require that any references acquired by a code is either generated due to an allocation or a function call; (b) they check bounds on array accesses, thus preventing stray memory accesses due to buffer overflows (and underflows); (c) they disallow accesses to null object, thus preventing applications from using undefined behavior to access memory that should be isolated; and (d) they ensure that all type casts are safe (and between compatible objects). Traditionally, languages providing these features (*e.g.,* Java, C#, Go, etc.) have been regarded as being too slow for systems programming.

This situation has improved with recent advances in language and runtime design, especially with the widespread adoption of LLVM as a common optimization backend for compilers. Furthermore, recent work has helped eliminate bounds checks in many common situations [2], and recently Intel has announced hardware support [18] to reduce the overhead of such checks. Finally, until recently most safe languages relied on garbage collection to safely allocate memory. The use of garbage collection results in occasional latency spikes which can adversely affect performance. However, recent languages such as Rust have turned to using reference counting (smart pointers) for heap allocations, leading to predictable latency for applications written in these languages. These developments prompted us to revisit the idea of software isolation for NFs; as we show later in §5, NetBricks achieves throughputs and 99^{th} percentile latency that is comparable with NFs written in more traditional system languages like C.

NFV requires more than just memory isolation; NFV must preserve the semantics of physical networks in the sense that an NF cannot modify a packet once it has been sent (we call this *packet isolation*). This is normally implemented by copying packets as they are passed from NF to NF, but this copying incurs a high performance overhead in packet-processing applications. We thus turn to unique types [13] to eliminate the requirement that packets be copied, while preserving packet isolation.

Unique types, which were originally proposed as a means to prevent data races, disallow two threads from simultaneously having access to the same data. They were designed so that this property could be statically verified at compile time, and thus impose no runtime overhead. We design NetBricks so that calls between NFs are marked to ensure that the sender looses access to the packet, ensuring that only a single NF has access to the packet. This allows us to guarantee that packet isolation holds without requiring any copying. Note that it is possible that some NFs (*e.g.,* IDSes or WAN optimizers) might require access to packet payloads after forwarding packets; in this case the NF is responsible for copying such data.

We refer to the combination of these techniques as Zero-Copy Soft Isolation (ZCSI), which is the cornerstone of NetBricks's execution environment. NetBricks runs as a single process, which maybe assigned one or more cores for processing and one or more NICs for packet I/O. We forward packets between NFs using function calls (*i.e.,* in most cases there are no queues between NFs in a chain, and queuing is done by the receiving NF).

Placement and Scheduling A single NetBricks process is used to run several NFs, which we assume are arranged in several parallel directed graphs – these parallel graphs would be connected to different network interfaces, as might be needed in a multi-tenant scenario where different tenants are handled by different chains of NFs. In addition to the nodes corresponding to the abstractions discussed above, these graphs have special nodes for receiving packets from a port, and sending packets out a NIC. Before execution NetBricks must decide what core is used to run each NF chain. Then, since at any time there can be several nodes in this graph with packets to process, NetBricks must make scheduling decisions about which packet to process next.

For placement, we envision that eventually external management systems (such as E2 [37]) would be responsible for deciding how NFs are divided across cores. At present, to maximize performance we place an entire NF chain on a single core, and replicate the processing graph across cores when scaling. More complex placement policies can be implemented using shuffle nodes, which allow packets to be forwarded across cores.

We use run-to-completion scheduling, *i.e.,* once a packet has entered the NF, we continue processing it until it exits. This then leaves the question of the order in which we let packets enter the NF, and how we schedule events that involve more than one packet. We denote such processing nodes as "schedulable", and these include nodes for receiving packets from a port, Window nodes (which need to schedule their UDF to run when enough data has been collected), and Group By nodes (which queue up packets to be processed by each of the branches). Currently, we use a round-robin scheduling policy to schedule among these nodes (implementing more complex scheduling is left to future work).

4 Implementation

While the previous section presented NetBricks's overall design, here we describe some aspects of its use and implementation.

4.1 Two Example NFs

We use two example NFs to demonstrate how NFs are written in NetBricks. First, in Listing 1 we present a trivial NF that decrements the IP time-to-live (TTL) field and drops any packets with TTL 0. NFs in NetBricks are generally packaged as public functions in a Rust module, and

```
1  pub fn ttl_nf<T: 'static + NbNode>(input: T)
2                              -> CompositionNode {
3    input.parse::<MacHeader>()
4         .parse::<IpHeader>()
5         .transform(box |pkt| {
6              let ttl = pkt.hdr().ttl() - 1;
7              pkt.mut_hdr().set_ttl(ttl);
8         })
9         .filter(box |pkt| {
10             pkt.hdr().ttl() != 0
11        })
12        .compose()
13 }
```

Listing 1: NetBricks NF that decrements TTL, dropping packets with TTL=0.

```
1  // cfg is configuration including
2  // the set of ports to use.
3  let ctx = NetbricksContext::from_cfg(cfg);
4  ctx.queues.map(|p| ttl_nf(p).send(p));
```

Listing 2: Operator code for using the NF in Listing 1

an operator can create a new instance of this NF using the `ttl_nf` function (line 1), which accepts as input a “source” node. The NF’s processing graph is connected to the global processing graph (*i.e.,* the directed graph of how processing is carried out end-to-end in a NetBricks deployment) through this node. The NF’s processing graph first parses the ethernet (MAC) header from the packet (line 3), and then parses the IP header (line 4). Note that in this case where the IP header begins depends on the contents of the ethernet header and can vary from packet to packet. Once the IP header has been parsed the NF uses the `transform` operator to decrement each packet’s TTL. Finally, we use the `filter` operator to drop all packets with TTL 0. The `compose` operator at the end of this NF acts as a marker indicating NF boundaries, and allows NFs to be chained together. This NF includes no `shuffle` operators, however by default NetBricks ensures that packets from the same flow are processed by a single core. This is to avoid bad interactions with TCP congestion control. Listing 2 shows how an operator might use this NF. First, we initialize a `NetbricksContext` using a user supplied configuration (Line 2). Then we create pipelines, such that for each pipeline (a) packets are received from an input queue; (b) received packets are processed using `ttl_nf`; and (c) packets are output to the same queue. Placement of each pipeline in this case is determined by the core to which a queue is affinitized, which is specified as a part of the user configuration.

Next, in Listing 3 we present a partial implementation of Maglev [9], a load balancer built by Google that was the subject of a NSDI 2016 paper. Maglev is responsible for splitting incoming user requests among a set of backend servers, and is designed to ensure that (a) it can be deployed in a replicated cluster for scalability and fault tolerance; (b) it evenly splits traffic between backends; and (c) it gracefully handles failures, both within the Maglev cluster and among the backends. Maglev uses a novel consistent hashing algorithm (based on a lookup table) to achieve these aims. It however needs to record the mapping between flows and backends to ensure that flows are not rerouted due to failures.

```
1  pub fn maglev_nf<T: 'static + NbNode>(
2                 input: T
3                 backends: &[str],
4                 ctx: nb_ctx,
5                 lut_size: usize)
6                 -> Vec<CompositionNode> {
7    let backend_ct = backends.len();
8    let lookup_table =
9        Maglev::new_lut(ctx,
10             backends,
11             lut_size);
12   let mut flow_cache =
13     BoundedConsistencyMap::<usize, usize>::new();
14
15   let groups =
16     input.shuffle(BuiltInShuffle::flow)
17          .parse::<MacHeader>()
18          .group_by(backend_ct, ctx,
19              box move |pkt| {
20                let hash =
21                  ipv4_flow_hash(pkt, 0);
22                let backend_group =
23                   flow_cache.entry(hash)
24                   .or_insert_with(|| {
25                  lookup_table.lookup(hash)});
26                backend_group
27      });
28     groups.iter().map(|g| g.compose()).collect()
29 }
```

Listing 3: Maglev [9] implemented in NetBricks.

The code in Listing 3 represents the packet processing and forwarding portions of Maglev; our code for generating the Maglev lookup table and consistent hashing closely resemble the pseudocode in Section 3.4 of the paper. The lookup table is stored in a bounded consistency state store, which allows the control plane to update the set of active backends over time. An instance of the Maglev NF is instantiated by first creating a Maglev lookup table (Line 8) and a cache for recording the flow to backend server mappings (Line 12). The latter is unsynchronized (*i.e.,* it is not shared across cores); this is consistent with the description in the Maglev paper. We then declare the NF (starting at line 15); we begin by using a shuffle node to indicate that the NF need all packets within a flow (line 16) to be processed by the same core, then parse the ethernet header, and add a group by node (Line 18). The group by node uses `ipv4_flow_hash`, a convenience function provided by NetBricks, to extract the flow hash (which is based on both the IP header and the TCP or UDP header of the packet) for the packet. This function is also responsible for ensuring that the packet is actually a TCP or UDP packet (the returned hash is 0 otherwise). The NF then uses this hash to either find the backend previously assigned to this flow (line 24) or assigns a new backend using the lookup table (line 25); this determines the group to which the packet being processed belongs. Finally, the NF returns a vector of composition nodes, where the n^{th} composition node corresponds to the n^{th} backend specified by the operator. The operator can thus forward traffic to each of the backends (or perform further processing) as appropriate. We compare the performance of the NetBricks version of Maglev to Google's reported performance in §5.2.

4.2 Operator Interface

As observed in the previous examples, operators running NetBricks chain NFs together using the same language (Rust) and tools as used by NF authors. This differs from current NF frameworks (*e.g.,* E2, OpenMANO, etc.) where operators are provided with an interface that is distinct from the language used to write network functions. Our decision to use the same interface is for two reasons: (a) it provides many optimization opportunities, in particular we use the Rust compiler's optimization passes to optimize the operator's chaining code, and can use LLVM's link-time optimization passes [28] to perform whole-program optimization, improving performance across the entire packet processing pipeline; and (b) it provides an easy means for operators to implement arbitrarily complicated NF chaining and branching.

4.3 Implementation of Abstractions

We now briefly discuss a few implementation details for abstractions in NetBricks. First, packet processing abstractions in NetBricks are lazy; *i.e.,* they do not perform computation until the results are required for processing. For example, parse nodes in NetBricks perform no computation until a transform, filter, group by, or similar node (*i.e.,* a node with a UDF that might access the packet header or payload) needs to process a packet. Secondly, as is common in high-performance packet processing, our abstractions process batches of packets at a time. Currently each of our abstractions implements batching to maximize common-case performance, in the future we plan on looking at techniques to choose the batching technique based on both the UDF and abstraction.

4.4 Execution Environment

The NetBricks framework builds on Rust, and we use LLVM as our runtime. We made a few minor modifications to the default Rust nightly install: we changed Cargo (the Rust build tool) to pass in flags that enabled machine specific optimizations and the use of vector instructions for fast memory access; we also implemented a Rust lint that detects the use of unsafe pointer arithmetic inside NFs, and in our current implementation we disallow building and loading of NF code that does not pass this lint. Beyond these minor changes, we found that we could largely implement our execution environment using the existing Rust toolchain. In the future we plan to use tools developed in the context of formal verification efforts like RustBelt [7] to (a) statically verify safety conditions in binary code (rather than relying on the Lint tool) and (b) eliminate more of the runtime checks currently performed by NetBricks.

5 Evaluation

5.1 Setup

We evaluate NetBricks on a testbed of dual-socket servers equipped with Intel Xeon E5-2660 CPUs, each of which has 10 cores. Each server has 128GB of RAM, which is divided equally between the two sockets. Each server is also equipped with an Intel XL710 QDA2 40Gb NIC. For our evaluation we disabled hyper-threading and adjusted the power settings to ensure that all cores ran at a constant 2.6GHz[2]. We also enabled hardware virtualization features including Intel VT. These changes are consistent with settings recommended for NFV applications. The servers run Linux kernel 4.6.0-1 and NetBricks uses DPDK version 16.04 and the Rust nightly version. For our tests we relied on two virtual switches (each configured as recommended by authors): OpenVSwitch with DPDK (OVS DPDK) [15], the de-facto virtual switch used in commercial deployments, and SoftNIC [17], a new virtual switch that has been specifically optimized for NFV use cases [37].

We run VMs using KVM; VMs connect to the virtual switch using DPDK's `vhost-user` driver. We run containers using Docker in privileged mode (as required by DPDK [8]), and connect them to the virtual switch using DPDK's ring PMD driver. By default, neither OpenVSwitch nor SoftNIC copy packets when using the ring PMD driver and thus do not provide packet isolation (because an NF can modify packets it has already sent). For most of our evaluation we therefore modify these switches to copy packets when connecting containers. However, even with this change, our approach (using DPDK's ring PMD driver) outperforms the commonly recommended approach of connecting containers with virtual switches using `veth` pairs (virtual ethernet devices that connect through the kernel). These devices entail a copy in the kernel, and hence have significantly worse performance than the ring based connections we use. Thus, the performance we report are a strict *upper bound* on can be achieved using containers safely.

For test traffic, we use a DPDK-based packet generator that runs on a separate server equipped with a 40Gb NIC and is directly connected to the test server without any in-

[2] In particular we disabled C-state and P-state transitions, isolated CPUs from the Linux scheduler, set the Linux CPU QoS feature to maximize performance, and disabled uncore power scaling.

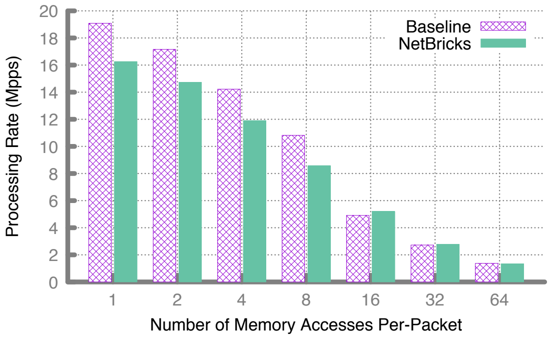

Figure 1: Throughput achieved by a NetBricks NF and an NF written in C using DPDK as the number of memory accesses in a large array grows.

tervening switches.The generator acts as traffic source and sink and we report the throughput and latency measured at the sink. For each run we measure the maximum throughput we can achieve with zero packet loss, and report the median taken across 10 runs.

5.2 Building NFs

5.2.1 Framework Overheads

We begin by evaluating the overheads imposed by NetBricks' programming model when compared to baseline NFs written more traditionally using C and DPDK. To ensure that we measure only framework overheads we configure the NIC and DPDK in an identical manner for both NetBricks and the baseline NF.

Overheads for Simple NFs As an initial sanity check, we began by evaluating overheads on a simple NF (Listing 1) that on receiving a packet, parses the packet until the IP header, then decrements the packet's IP time-to-live (TTL) field, and drops any packets whose TTL equals 0. We execute both the NetBricks NF and the equivalent C application on a single core and measure throughput when sending 64 byte packets. As expected, we find that the performance for the two NFs is nearly identical: across 10 runs the median throughput for the native NF is 23.3 million packet per-second, while NetBricks achieves 23.2 million packets per second. In terms of latency, at 80% load, the 99^{th} percentile round trip time for the native NF is 16.15µs, as compared to 16.16µs for NetBricks.

Overheads for Checking Array Bounds Our use of a safe language imposes some overheads for array accesses due to the cost of bounds checking and such checks are often assumed to be a dominant source of overhead introduced by safe languages.[3] While these checks can be eliminated statically in some cases (*e.g.,* where bounds can be placed on the index statically), this is not always possible. We measured the impact of these checks using a network function that updates several cells in a 512KB array while processing each packet. The set of cells to be updated depends on the UDP source port number of the packet being processed, making it impossible to eliminate array bounds checks. We compare the overheads for our implementation in NetBricks to a baseline NF written in C (using DPDK for packet I/O), and behaving identically. In both cases we use a single-core and use packets with randomly assigned UDP source ports. Figure 1 shows the throughput achieved by each NF as the number of memory accesses per packet is increased. When processing a packet necessitates a single memory access, NetBricks imposes a 20% overhead compared to the baseline. We see that this performance overhead remains for a small number (1-8) of accesses per packet. However, somewhat counter-intuitively, with 16 or higher accesses per packet, the performance overhead of our approach drops; this is because, at this point, the number of cache misses grows and the performance impact of these misses dominates that from our bounds checks.

To test the impact of this overhead in a more realistic application we implemented a longest prefix match (LPM) lookup table using the DIR-24-8 algorithm [16] in Rust, and built a NetBricks NF that uses this data structure to route IP packets. We compare the performance of this NF to one implemented in C, which uses the DIR-24-8 implementation included with DPDK [24]. Lookups using this algorithm require between 1 and 2 array accesses per packet. For our evaluation we populated this table with 16000 random rules. We find that NetBricks can forward 15.73 million packet per second, while the native NF can forward 18.03 million packets per second (so the NetBricks NF is 14% slower). We also measure the 99^{th} percentile round trip time at 80% load (*i.e.,* the packet generator was generating traffic at 80% of the 0-loss rate), this value indicates the per-packet latency for the NF being tested. The $99th$ percentile RTT for NetBricks was 18.45µs, while it was 16.15µs for the native NF, which corresponds to the observed difference in throughputs.

5.2.2 Generality of Programming Abstractions

To stress test the generality of our programming abstractions, we implemented a range of network functions from the literature:

- Firewall: is based on a simple firewall implemented in Click [5]; the firewall performs a linear scan of an access control list to find the first matching entry.
- NAT: is based on MazuNAT [41] a Click based NAT implemented by Mazu Networks, and commonly used in academic research.

[3] Null-checks and other safety checks performed by the Rust runtime are harder to separate out; however, these overheads are reflected in the overall performance we report below.

NF	NetBricks	Baseline
Firewall	0.66	0.093
NAT	8.52	2.7
Sig, Matching	2.62	0.983
Monitor	5	1.785

Table 1: Throughputs for NFs implemented using NetBricks as compared to baseline from the literature.

- Signature Matching: a simple NF similar to the core signature matching component of the Snort intrusion prevention system [42].
- Monitor: maintains per-flow counters similar to the monitor module found in Click and commonly used in academic research [43]
- Maglev: as described in § 4, we implemented a version of the Maglev scalable load-balancer design [9].

In Table 1, we report the per-core throughput achieved by the first four applications listed above, comparing our NetBricks implementation and the original system on which we based on our implementation. We see that our NetBricks implementations often outperform existing implementations – *e.g.,* our NAT has approximately 3× better performance than MazuNAT [41]. The primary reason for this difference is that we incorporate many state-of-the-art optimizations (such as batching) that were not implemented by these systems.

In the case of Maglev, we do not have access to the source code for the original implementation and hence we recreate a test scenario similar to that corresponding to Figure 9 in [9] which measures the packet processing throughput for Maglev with different kinds of TCP traffic. As in [9], we generate short-lived flows with an average of 10 packets per flow and use a table with 65,537 entries (corresponding to the small table size in [9]). Our test server has a 40Gbps link and we measure throughput for (min-size) 64B packets. Table 2 shows the throughput achieved by our NetBricks implementation for increasing numbers of cores (in Mpps), together with comparable results reported for the original Maglev system in [9]. We see that our NetBricks implementation offers between 2.9× and 3.5× better performance than reported in [9]. The median latency we observed in this case was 19.9μS while 99th percentile latency was 32μS. We note however that (a) we ran on different hardware; and (b) we did not have access to the base implementation and hence comment on parity. Therefore these numbers are not meant to indicate that our performance is better, just that NetBricks can achieve comparable results as obtained by a hand tuned NF.

# of Cores	NetBricks Impl.	Reported
1	9.2	2.6
2	16.7	5.7
3	24.5	8.2
4	32.24	10.3

Table 2: Throughputs for the NetBricks implementation of Maglev (NetBricks) when compared to the reported throughput in [9] (Reported) in millions of packets per second (MPPS).

Our point here is not that NetBricks will outperform highly optimized native implementations; instead, our results merely suggest that NetBricks can be used to implement a wide variety of NFs, and that these implementations are both simpler than the native implementations (*e.g.,* our Maglev implementation is 150 lines of code) and roughly comparable in performance.

5.3 Execution Environment

NetBricks exploits the isolation properties of safe languages and runtime checks to avoid the costs associated with crossing process and/or core boundaries. We first quantify these savings in the context of a single NF and then evaluate how these benefits accrue as the length of a packet's NF chain increases. Note that these crossing costs are only important for simple NFs; once the computational cost of the NF becomes the bottleneck, then our execution environment becomes less important (though NetBricks's ability to simply implement high-performance NFs becomes more important).

5.3.1 Cost of Isolation: Single NF

We evaluate the overhead of using VMs or containers for isolation and compare the resultant performance to that achieved with NetBricks. We first consider the simplest case of running a single test NF (which is written using NetBricks) that swaps the source and destination ethernet address for received packets and forwards them out the same port. The NetBricks NF adds no additional overhead when compared to a native C NF, and running the same NF in all settings (VM, containers, NetBricks) allows us to focus on the cost of isolation.

The setup for our experiments with containers and VMs is shown in Figure 2: a virtual switch receives packets from the NIC, these packets are then forwarded to the NF which is running within a VM or container. The NF processes the packet and sends it back to the vSwitch, which then sends it out the physical NIC. Our virtual switches and NFs run on DPDK and rely on polling. We hence assign each NF its own CPU core and assign two cores to the switch for polling packets from the NIC and the container.[4] Isolation introduces two sources of overheads: overheads from cache and context switching costs associated with crossing process (and in our case core) boundaries, and overheads

[4] This configuration has been shown to achieve better performance than one in which the the switch and NFs share a core [35]. Our own experiments confirm this, we saw as much as 500% lower throughput when cores were shared.

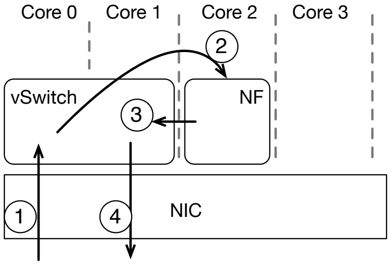

Figure 2: Setup for evaluating single NF performance for VMs and containers.

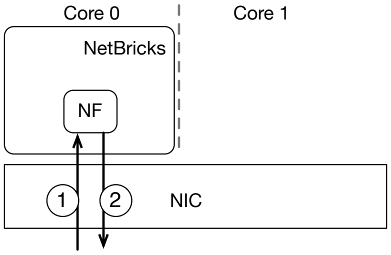

Figure 3: Setup for evaluating single NF performance using NetBricks.

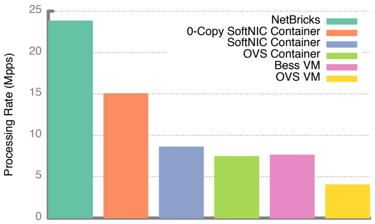

Figure 4: Throughput achieved using a single NF running under different isolation environments.

from copying packets. To allow us to analyze these effects separately we include in our results a case where SoftNIC is configured to send packets between containers without copying (0-copy SoftNIC Container), even though this violates our desired packet isolation property. We compare these results to NetBricks running in the setup shown in Figure 3. In this case NetBricks is responsible for receiving packets from the NIC, processing them using the NF code and then sending them back out. We run NetBricks on a single core for this evaluation.

Figure 4 shows the throughput achieved for the different isolation scenarios when sending 64B minimum sized packets. Comparing the 0-copy SoftNIC throughput against NetBricks's throughput, we find that just crossing cores and process isolation boundaries results in performance degradation of over 1.6× when compared to NetBricks (this is despite the fact that our NetBricks results used fewer cores overall; 1 core for NetBricks vs. 3 in the other cases). When packets are copied (SoftNIC Container) throughput drops further and is 2.7× worse than NetBricks. Generally the cost for using VMs is higher than the cost for using Containers; this is because Vhost-user, a virtualized communication channel provided by DPDK for communicating with VMs imposes higher overheads than the ring based communication channel we use with containers.

The previous results (Figure 9) focused on performance with 64B packets, and showed that as much as 50% of the overhead in these systems might be due to copying packets. We expect that this overhead should increase with larger packets, hence we repeated the above tests for 1500B packets and found that the per-packet processing time (for those scenarios that involve copying packets) increased by approximately 15% between 64B and 1500B packets (the small size of the increase is because the cost of allocation dominates the cost of actually copying the bits).

5.3.2 Cost of Isolation: NF Chains

Next, we look at how performance changes when each packet is handled by a *chain* of NFs. For simplicity, we generate chains by composing multiple instances of a single test NF; *i.e.,* every NF in the chain is identical and we only vary the length of the chain. Our test NF performs the following processing: on receiving a packet, the NF parses the ethernet and IP header, and then decrement the time-to-live (TTL) field in the IP header. The NF drops any packets where the TTL is 0.

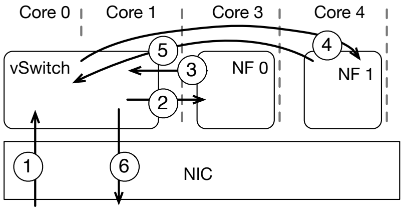

Figure 5: Setup for evaluating the performance for a chain of NFs, isolated using VMs or Containers.

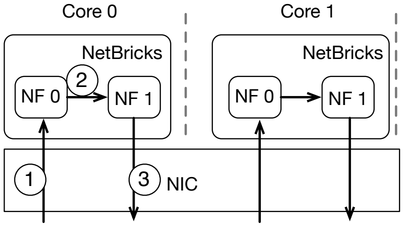

Figure 6: Setup for evaluating the performance for a chaining of NFs, running under NetBricks.

We use the setup shown in Figure 5 to measure these overheads when using VMs and containers. As before, we assign the virtual switch two cores, and we place each VM or container on a separate core. We evaluate NetBricks using the setup shown in Figure 6. We ran NetBricks in two configurations: (a) one where NetBricks was run on a single core, and (b) another where we gave NetBricks as many cores as the chain length; in the later case NetBricks uses as many cores as the container/VM runs.

In Figure 7 we show the throughput as a function of increasing chain length. We find that NetBricks is up to 7× faster than the case where containers are connected using SoftNIC and up to 11× faster than the case where VMs are connected using SoftNIC. In fact NetBricks is faster

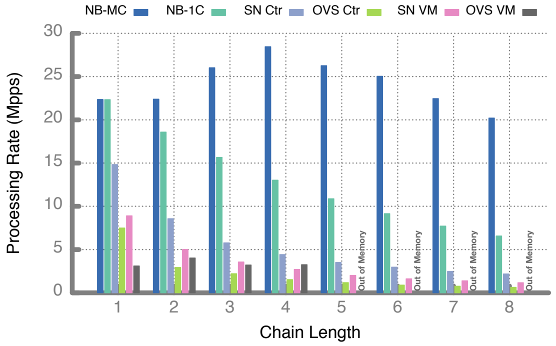

Figure 7: Throughput with increasing chain length when using 64B packets. In this figure NB-MC represents NetBricks with multiple cores, NB-1C represents NetBricks with 1 core.

even when run on a single core, we observe that it provides 4× higher throughput than is achieved when containers are connected through SoftNIC, and up to 6× higher throughput when compared to the case where VMs are connected using SoftNIC. Furthermore, by comparing to the 0-copy SoftNIC case, we find that for 64B packets copying can result in a performance drop of up to 3×. Finally, observe that there is a dip in NetBricks's performance with multiple cores once the chain is longer than four elements. This is because in our setup I/O becomes progressively more expensive as more cores access the same NIC, and with more than 4 parallel I/O threads this cost dominates any improvements from parallelism. We believe this effect is not fundamental, and is a result of our NIC and the current 40Gbps driver in DPDK. NetBricks's performance benefits are even higher when we replace SoftNIC with OpenVSwitch.[5]

The above results are for 64B packets; as before, we find that while copying comes at a large fixed cost (up to 3× reduction in throughput), increasing packet sizes only results in an approximately 15% additional degradation. Finally, we also measured packet processing latency when using NetBricks, containers and VMs; Figure 8 shows the 99^{th} percentile round trip time at 80% of the maximum sustainable throughput as a metric for latency.

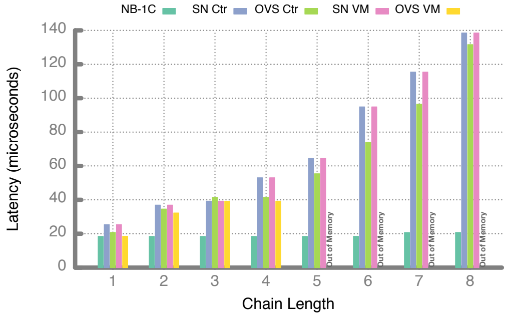

Figure 8: 99^{th} percentile RTT for 64B packets at 80% load as a function of chain length.

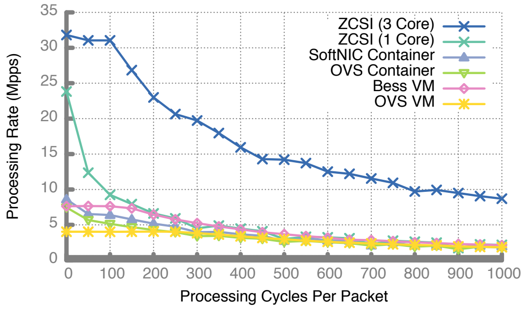

Figure 9: Throughput for a single NF with increasing number of cycles per-packet using different isolation techniques.

Effect of Increasing NF Complexity Finally, we analyze the importance of our techniques for more complex NFs. We use cycles required for processing each packet as a proxy for NF complexity. We reuse the setup for single NF evaluations (Figure 2, 3), but modify the NF so that it busy loops for a given number of cycles after modifying the packet, allowing us to vary the per-packet processing time. Furthermore, note that in the case where VMs or containers the setup itself uses 3 cores (1 for the NF and 2 for the vSwitch). Therefore, for this evaluation, in addition to measuring performance with NetBricks on 1 core, we also measure performance when NetBricks is assigned 3 cores (equalizing resources across the cases).

Figure 9 shows the throughput as a function of per-packet processing requirements (cycles). As expected, as we increase NF complexity, packet processing time starts to be the dominant factor for determining performance, and our runtime improvements have minimal effect once an NF needs more than 300 cycles per packet. This reduction in benefits when NF processing demands dominate also applies to fast packet processing libraries such as DPDK. Note however that the gains when NetBricks is given as many cores as the traditional approaches (three) continue to be significant even when NFs need more than 1000 cycles per packet. Thus, NetBricks' approach to isolation provides better performance per unit of allocated resource when compared to current approaches.

6 Related Work

The works most closely related to NetBricks' programming model are Click and Snabb switch [14]. We have compared NetBricks and Click throughout the paper, do not provide further discussion here. Recent extensions to Click, *e.g.,* NBA [26] and ClickNP [29], have looked at how to implement optimized Click elements through the use of GPUs (NBA) and FPGAs (ClickNP). While offloading function-

[5] We were unable to run experiments with more than four VMs chained together using OpenVSwitch because we ran out of memory in our configuration.

Framework	Memory Isolation	Packet Isolation	Overheads
xOMB [1]	✗	✗	Low (function call)
CoMB [43]	✗	✗	Low (function call)
NetVM [21]	✓	✗	Very high (VM)
ClickOS [32]	✓	✓	High (lightweight VM)
HyperSwitch [40]	✓	✓	Very high (VM)
mSwitch [19]	✓	✓	Very high (VM)
NetBricks	✓	✓	Low (function call)

Table 3: A comparison with other NFV frameworks.

ality to such devices can yield great performance improvements, this is orthogonal to our work. Adding the use of offloads in NetBricks is left to future work. Snabb provides the same programming model as Click but uses Lua [22] instead of C++ for programming, which allows the use of a high-level language but without actually raising the level of abstraction (in terms of having the programmer deal with all the low-level packet-handling issues).

There has also been a long line of work on developing network applications on specialized networking hardware including NPUs [46], FPGAs [33] and programmable switches [4]. Recent work including P4 [3] and Packet Transactions [45] have looked at providing high level programming tools for such hardware. Our work focuses on network programming for general purpose CPUs and is complementary to this work.

In terms of NetBricks' execution model, work on library operating systems (*e.g.,* MirageOS [31] and Drawbridge [39]) has decreased VM resource overheads and improved VM performance by reducing the amount of code run within each VM and improving the hypervisor. While these projects have provided substantial performance improvements, they do not eliminate the isolation overheads we focus on here nor do they address how to perform efficient I/O in this environment.

As we have noted previously, our execution model is closely related to software-isolated processes (SIPs) proposed by Singularity [20]. The main difference is that our work focuses on a single application domain – network functions – where inter-NF communication is common and greatly benefits from the use of software isolation. Furthermore, Singularity was designed as a general purpose, microkernel-based operating system, and focused on providing an efficient general implementation for application developers. As a result Singularity's design choices – *e.g.,* the use of a garbage collected language, communication through an exchange heap and queued channel, etc. – are not optimized for the NFV use case.

Other work has proposed a variety of execution frameworks specific to NFV [1,19,21,32,40,43]. We can broadly divide these frameworks into two groups: Click-like frameworks that run all NFs in a single process without isolation, and VM-based frameworks. We present a comparison of these frameworks and NetBricks in Table 3. As shown, only NetBricks provides both isolation and low overheads. Finally, In-Net [47] has looked at providing traffic isolation for NFs in a network, and is orthogonal to our work.

Several attempts have also been made to offload vSwitch functionality to NIC hardware. For example, FasTrak [34] advocates using hardware virtualization (SR-IOV [6]) and built-in switching capabilities of commodity NICs to interconnect VMs. This approach eliminates the cost of copying packets in software by using hardware DMA. However, I/O bus bandwidth is an order-of-magnitude lower (a few GB/s) than cache and memory bandwidth (10s-100s of GB/s), and this limits the number of packets that can be transmitted in parallel and thus reduces the throughput that can be achieved. Offloading switching to hardware also limits flexibility in how packets are steered across NFs; *e.g.,* Intel's 10 G NICs only support basic L2 switching.

IO-Lite [36], Container Shipping [38], and work done for Solaris [25] have looked at solutions for implementing zero-copy I/O. IO-Lite provided zero-copy isolation by making buffers immutable. This necessitates creating a new buffer on any write (similar to copy-on-write techniques) and would therefore incur performance degradation when modifications are required. Container shipping and the Solaris approach unmap pages from the sending process to provide zero-copy isolation. Page table modifications require a trap into the kernel, and come at a significant penalty [48]. By contrast our implementation of 0-copy I/O imposes no runtime overheads.

7 Conclusion

As can be seen from our brief review of related work, NetBricks is the only approach that enables developers to write in high-level abstractions (thereby easing development) while maintaining good performance and memory/-packet isolation. We are continuing to explore the limits of NetBricks's generality – by implementing new NFs – and increase the range of NetBricks's low-level optimizations, some of which are currently rather primitive. In service of these goals, we have also made NetBricks and our examples available to the community at netbricks.io.

8 Acknowledgment

We thank our shepherd George Porter and the anonymous reviewers for their comments. We also thank Ion Stoica, Amin Tootoonchian and Shivaram Venkatraman for their helpful feedback, which influenced both the design of our system and the contents of this paper. This work was funded in part by a grant from Intel Corporation, and by NSF awards 1216073 and 1420064.

References

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *ANCS*, 2012.

[2] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI*, 2000.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] 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*, 2013.

[5] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[6] Y. Dong, X. Yang, L. Xiaoyong, J. Li, H. Guan, and K. Tian. High Performance Network Virtualization with SR-IOV. In *IEEE HPCA*, 2012.

[7] D. Dreyer. RustBelt: Logical Foundations for the Future of Safe Systems Programming. http://plv.mpi-sws.org/rustbelt/ (Retrieved 05/05/2016), 2015.

[8] J. Eder. Can you run DPDK in a Container. Redhat Blog http://goo.gl/UBdZpL, 2015.

[9] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.

[10] ETSI. Network Functions Virtualisation. Retrieved 07/30/2014 http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[11] L. Foundation. OPNFV. https://www.opnfv.org/, 2016.

[12] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. http://arxiv.org/abs/1305.0209, 2013.

[13] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, 2012.

[14] L. Gorrie. SNABB Switch. https://goo.gl/8ox9kE retrieved 07/16/2015.

[15] J. Gross. The Evolution of OpenVSwitch. http://goo.gl/p7QVek retrieved 07/13/2015, 2014. Talk at LinuxCon.

[16] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM*, 1998.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[18] D. Hansen. Intel Memory Protection Extensions (Intel MPX) for Linux*. https://01.org/blogs/2016/intel-mpx-linux retrieved 05/07/2016, 2016.

[19] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mSwitch: A Highly-Scalable, Modular Software Switch. In *SOSR*, 2015.

[20] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[21] J. Hwang, K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. *Network and Service Management, IEEE Transactions on*, 12(1):34–47, 2015.

[22] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes Filho. Lua–an Extensible Extension Language. In *Software: Practice & Experience*, 1995.

[23] Intel. Data Plane Develpment Kit. http://dpdk.org/, 2016.

[24] Intel. DPDK: rte_table_lpm.h reference. http://goo.gl/YBS4UO retrieved 05/07/2016, 2016.

[25] Y. A. Khalidi and M. N. Thadani. An Efficient Zero-Copy I/O Framework for Unix. *Sum Mircrosystems Laboratories, Inc. Tech Report*, 1995.

[26] J. Kim, K. Jang, K. Lee, S. Ma, J. Shim, and S. B. Moon. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *EuroSys*, 2015.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*. IEEE, 2004.

[29] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In *SIGCOMM*, 2016.

[30] D. Lopez. OpenMANO: The Dataplane Ready Open Source NFV MANO Stack. In *IETF Meeting Proceedings, Dallas, Texas, USA*, 2015.

[31] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

[32] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[33] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *Workshop on Programmable routers for extensible services of tomorrow*, 2008.

[34] R. Niranjan Mysore, G. Porter, and A. Vahdat. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *CoNEXT*, 2013.

[35] OpenVSwitch. Using Open vSwitch with DPDK. https://github.com/openvswitch/ovs/blob/master/INSTALL.DPDK.md, 2016.

[36] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, 2000.

[37] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[38] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating System Support for I/O-intensive Applications. *Computer*, 27(3):84–93, 1994.

[39] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *ASPLOS*, 2011.

[40] K. K. Ram, A. L. Cox, M. Chadha, S. Rixner, T. W. Barr, R. Smith, and S. Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *USENIX ATC*, 2013.

[41] Riverbed. Mazu Networks. http://goo.gl/Y6aeEg, 2011.

[42] M. Roesch et al. Snort: Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[43] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[44] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[45] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown. Packet Transactions: High-level Programming for Line-Rate Switches. In *SIGCOMM*, 2016.

[46] H. Song. Protocol-Oblivious Forwarding: Unleash the Power of SDN Through a Future-Proof Forwarding Plane. In *HotSDN*, 2013.

[47] R. Stoenescu, V. A. Olteanu, M. Popovici, M. Ahmed, J. Martins, R. Bifulco, F. Manco, F. Huici, G. Smaragdakis, M. Handley, and C. Raiciu. In-Net: In-Network Processing for the Masses. In *EuroSys*, 2015.

[48] L. Torvalds. Linux Page Fault Daemon Performance. Google+ https://plus.google.com/+LinusTorvalds/posts/YDKRFDwHwr6, 2014.