A New Approach to Network Functions

Aurojit Panda
Current Approach to NFV
Current Approach to NFV

Firewall  Cache  
NFs written by experts shipped as VM or container.
Current Approach to NFV

NFs written by experts shipped as VM or container.

NFs built to target hardware & software features.

```
static int _xbegin(void) {
    int ret = XBEGIN_STARTED;
    asm volatile(".byte 0xc7,0xf8 ; .long 0" :
               "+a" (ret)
               :: "memory");
    return ret;
}

static void _xend(void) {
    asm volatile(".byte 0x0f,0x01,0xd5"
               :: "memory");
}
```
Current Approach to NFV

- Firewall
- Cache

NFs written by experts shipped as VM or container.

- vSwitch
- Firewall
- Cache

Executed in VMs or containers for isolation.

- FTMB
- FPGA
- AVX-512
- TSX

NFs built to target hardware & software features.
Current Approach to NFV

NFs written by experts shipped as VM or container.

NFs built to target hardware & software features.

Executed in VMs or containers for isolation.

Core placement for performance.
Problems with the Current Approach

- High overheads for isolation and chaining.
High Overheads for Isolation

![Bar chart showing high overheads for isolation.

- Processing Rate (Mpps):
  - No Isolation: 25 Mpps]
High Overheads for Isolation

![Graph showing processing rate comparison between No Isolation and OVS VM.](image)

- **No Isolation**: High processing rate (30 Mpps)
- **OVS VM**: Much lower processing rate (4 Mpps)
High Overheads for Isolation

![Graph showing processing rates with different isolation methods.](chart)

- **No Isolation**: High processing rate
- **OVS VM**: Moderate processing rate
- **BESS VM**: Low processing rate
High Overheads for Isolation

![Bar Chart]

- No Isolation
- OVS VM
- BESS VM
- BESS Container

Processing Rate (Mpps)
High Overheads for Chaining

![Bar chart showing processing rate (Mpps) vs chain length for containers and VMs.]
Problems with the Current Approach

• High overheads for isolation and chaining.

• Hard to write high-performance network functions.
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++. 

Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
  - Simplify batched I/O of packets, provide some common data structures.
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
  - Simplify batched I/O of packets, provide some common data structures.
- **Programmers** responsible for meeting NF **performance requirements**.
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
  - Simplify batched I/O of packets, provide some common data structures.
- **Programmers** responsible for meeting NF performance requirements.
  - Write code to maintain access locality, prevent pipeline stalls, etc.
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
  - Simplify batched I/O of packets, provide some common data structures.
- **Programmers** responsible for meeting NF performance requirements.
  - Write code to maintain access locality, prevent pipeline stalls, etc.
- Result: *Largely written by the same companies that built middleboxes.*
Hard to Write New NFs

- NFs written in low-level languages, e.g., C or C++.
- Make use of packet processing and I/O libraries like DPDK and netmap.
  - Simplify batched I/O of packets, provide some common data structures.
- **Programmers** responsible for meeting NF **performance requirements**.
  - Write code to maintain access locality, prevent pipeline stalls, etc.
- Result: *Largely written by the same companies that built middleboxes.*
- Hard for **carriers** or **new entrants** to develop NFs, limiting innovation.
Problems with the Current Approach

- High overheads for isolation and chaining.
- Hard to write high-performance network functions.
- Hard to upgrade existing network functions to utilize new features.
Hard to Upgrade Existing NFs

• Rapidly evolving set of **hardware accelerators** including:
Hard to Upgrade Existing NFs

• Rapidly evolving set of **hardware accelerators** including:
  
  • For example AVX, transactional memory (TSX), AES-NI, FPGAs, etc.
Hard to Upgrade Existing NFs

• Rapidly evolving set of **hardware accelerators** including:
  
  • For example AVX, transactional memory (TSX), AES-NI, FPGAs, etc.

• Rapidly evolving software architecture within which NFs are deployed:
Hard to Upgrade Existing NFs

- Rapidly evolving set of **hardware accelerators** including:
  - For example AVX, transactional memory (TSX), AES-NI, FPGAs, etc.
- Rapidly evolving software architecture within which NFs are deployed:
  - State management for fault tolerance/scaling, enhanced scheduling, etc.
Hard to Upgrade Existing NFs

• Rapidly evolving set of **hardware accelerators** including:
  • For example AVX, transactional memory (TSX), AES-NI, FPGAs, etc.
• Rapidly evolving software architecture within which NFs are deployed:
  • State management for fault tolerance/scaling, enhanced scheduling, etc.
• Feature availability dictated by deployment environment, use dictated by vendor.
Hard to Upgrade Existing NFs

• Rapidly evolving set of **hardware accelerators** including:
  • For example AVX, transactional memory (TSX), AES-NI, FPGAs, etc.

• Rapidly evolving software architecture within which NFs are deployed:
  • State management for fault tolerance/scaling, enhanced scheduling, etc.
  • Feature availability dictated by deployment environment, use dictated by vendor.

• Result: *Delays before new features are used, increased cost for upgrades.*
Problems with the Current Approach

- High overheads for isolation and chaining.

- Hard to write high-performance network functions.

- Hard to upgrade existing network functions to utilize new features.
NetBricks Addresses these Problems
What is NetBricks

- A new execution environment and programming framework for NFs.
What is NetBricks

• A new execution environment and programming framework for NFs.

• Up to an order of magnitude better NF performance.
What is NetBricks

• A new execution environment and programming framework for NFs.

• Up to an order of magnitude better NF performance.

• Open source project. Currently developed and maintained by NEFELI NETWORKS.
NetBricks Overview
NetBricks Overview

• Compile time type checks with minimal runtime checks for isolation.

• Same guarantees as existing approaches.

• Significantly lower overheads for isolation and chaining.
NetBricks Overview

• Compile time type checks with minimal runtime checks for isolation.

• Same guarantees as existing approaches.

• Significantly lower overheads for isolation and chaining.

• High-level dataflow model for expressing NF functionality.
NetBricks Overview

• Compile time type checks with minimal runtime checks for isolation.
• Same guarantees as existing approaches.
• Significantly lower overheads for isolation and chaining.
• High-level dataflow model for expressing NF functionality.
• NFs built using framework defined operators and user defined functions.
NetBricks Overview

• Compile time type checks with minimal runtime checks for isolation.

• Same guarantees as existing approaches.

• Significantly lower overheads for isolation and chaining.

• High-level dataflow model for expressing NF functionality.

• NFs built using framework defined operators and user defined functions.

• NF can be expressed simply and succinctly.
NetBricks Overview

• Compile time type checks with minimal runtime checks for isolation.
• Same guarantees as existing approaches.
• Significantly lower overheads for isolation and chaining.
• High-level dataflow model for expressing NF functionality.
• NFs built using framework defined operators and user defined functions.
• NF can be expressed simply and succinctly.
• Operator provided by framework, simplify NF upgrades.
NetBricks Overview

- Compile time type checks with minimal runtime checks for isolation.
- Same guarantees as existing approaches.
- Significantly lower overheads for isolation and chaining.
- High-level dataflow model for expressing NF functionality.
- NFs built using framework defined operators and user defined functions.
- NF can be expressed simply and succinctly.
- Operator provided by framework, simplify NF upgrades.
NetBricks: Execution Environment
Execution Environment Requirements

- Execution environment must satisfy three requirements
Execution Environment Requirements

- **Execution environment** must satisfy three requirements
  - Performance: Process packets at line rate.
Execution Environment Requirements

- **Execution environment** must satisfy three requirements:
  
  - Performance: Process packets at line rate.
  
  - Consolidation: Maximize number of NFs that can be consolidated.
Execution Environment Requirements

- **Execution environment** must satisfy three requirements
  - Performance: Process packets at line rate.
  - Consolidation: Maximize number of NFs that can be consolidated.
  - Isolation: Ensure NFs do not affect each other
Isolation
Isolation

- **Memory Isolation**: Partition memory spatially between NFs.
Isolation

- **Memory Isolation**: Partition memory spatially between NFs.

- **Packet Isolation**: Partition packet memory temporally between NFs.
Isolation

- **Memory Isolation**: Partition memory spatially between NFs.

- **Packet Isolation**: Partition packet memory temporally between NFs.

- **Performance Isolation**: One NF does not affect another’s performance.
Isolation

- **Memory Isolation**: Partition memory spatially between NFs.

- **Packet Isolation**: Partition packet memory temporally between NFs.

- **Performance Isolation**: One NF does not affect another’s performance.

- Ongoing work: Partition last level cache.
Why Isolation?

• Enables consolidation in NFV deployments.
Why Isolation?

• Enables consolidation in NFV deployments.

• NFs can be run by different tenants, built by different vendors.
Why Isolation?

- Enables **consolidation** in NFV deployments.
- NFs can be run by different **tenants**, built by different **vendors**.
- Enables **consolidation** for NFs like SSL proxies that have **secrets**.
Why Isolation?

- Enables **consolidation** in NFV deployments.
  - NFs can be run by different **tenants**, built by different **vendors**.
- Enables **consolidation** for NFs like SSL proxies that have **secrets**.
- Isolation is a building block for protecting secrets in applications.
Why Isolation?

- Enables **consolidation** in NFV deployments.
  - NFs can be run by different **tenants**, built by different **vendors**.
- Enables **consolidation** for NFs like SSL proxies that have **secrets**.
- Isolation is a building block for protecting secrets in applications.
- **Simplifies programming**: don’t need to worry about other **programs** and **NFs**.
Why Isolation?

- Enables **consolidation** in NFV deployments.
  - NFs can be run by different **tenants**, built by different **vendors**.
- Enables **consolidation** for NFs like SSL proxies that have **secrets**.
- Isolation is a building block for protecting secrets in applications.
- **Simplifies programming**: don’t need to worry about other **programs** and **NFs**.
- Lack of isolation between drivers is a major cause of crashes in OSes.
Memory Isolation
Packet Isolation
Performance
Memory Isolation
Packet Isolation
Performance
Memory Isolation
Packet Isolation
Performance
Memory Isolation

Packet Isolation

Performance
vSwitch

NIC

VM/Container

NIC

VM/Container

VM/Container

✔ Memory Isolation
Packet Isolation
Performance
Memory Isolation
Packet Isolation
Performance
Memory Isolation
Packet Isolation
Performance
Memory Isolation

Packet Isolation

Performance
Memory Isolation ✔
Packet Isolation ✔
Performance ✗
NetBricks: Low Overhead Isolation

![Graph showing processing rate comparison between different isolation methods: No Isolation, NetBricks, OVS VM, BESS VM, BESS Container.](image)
NetBricks: Key Insight

• Runtime mechanisms too expensive for NFV workloads.
NetBricks: Key Insight

• **Runtime mechanisms** too expensive for NFV workloads.

• Process a packet approximately every 100ns (10 MPPS) or faster.
NetBricks: Key Insight

• **Runtime mechanisms** too expensive for NFV workloads.
  • Process a packet approximately every 100ns (10 **MPPS**) or faster.

• Must rely on **static** compile-time checks for **isolation**.
Memory Isolation at Compile Time

• Types and runtime checks can provide memory isolation within a process.
Memory Isolation at Compile Time

- Types and runtime checks can provide memory isolation within a process.
- Check isolation almost entirely at compile time, limited runtime overhead.
Memory Isolation at Compile Time

• Types and runtime checks can provide memory isolation within a process.

• Check isolation almost entirely at compile time, limited runtime overhead.

• Built on Rust - type checks, bound checks, no garbage collection.
Memory Isolation at Compile Time

• Types and runtime checks can provide memory isolation within a process.

• Check isolation almost entirely at compile time, limited runtime overhead.

• Built on Rust - type checks, bound checks, no garbage collection.

• Framework designed to meet the rest of the memory isolation requirements.
Approaches to Packet Isolation

- Packets are passed between network functions - memory isolation insufficient.
- Existing approaches convert this temporal problem to a spatial one.
- Copy packets from one packet space to another.
Approaches to Packet Isolation

- Packets are passed between network functions - memory isolation insufficient.

- Existing approaches convert this temporal problem to a spatial one.

- Copy packets from one packet space to another.
Approaches to Packet Isolation

- Packets are passed between **network functions** - memory isolation insufficient.
- Existing approaches convert this **temporal problem** to a **spatial one**.
- **Copy packets** from one packet space to another.
Linear Types: Packet Isolation at Compile Time

• Solution: Use **linear types** (1990s) for isolation.
Linear Types: Packet Isolation at Compile Time

• Solution: Use **linear types** (1990s) for isolation.

• Syntax marks argument that are **moved**.

```rust
fn consume(a: Packet) {
    // Work with packet.
}
```
Linear Types: Packet Isolation at Compile Time

- Solution: Use **linear types** (1990s) for isolation.
- Syntax marks argument that are **moved**.
  
  **Argument** moved during calls.

```rust
fn consume(a: Packet) {
    // Work with packet.
}

// pkt is a packet
consume(pkt);
```
Linear Types: Packet Isolation at Compile Time

- Solution: Use **linear types** (1990s) for isolation.
- Syntax marks argument that are **moved**.
  - **Argument** moved during calls.
  - Ownership is transferred to callee.

```rust
fn consume(a: Packet) {
    // Work with packet.
}

// pkt is a packet
consume(pkt);
```
Linear Types: Packet Isolation at Compile Time

- Solution: Use **linear types** (1990s) for isolation.

- Syntax marks argument that are moved.

- **Argument** moved during calls.
  - Ownership is transferred to callee.

- **Moved variables** can not be reused.

```r
fn consume(a: Packet) {
    // Work with packet.
}

// pkt is a packet
consume(pkt);

pkt.set_length(200)
```
NetBricks: Packet Isolation
NetBricks: Packet Isolation

 NF A  Packet  NF B
NetBricks: Packet Isolation
NetBricks: Packet Isolation

- Linear types implemented by Rust for concurrency.
NetBricks: Packet Isolation

- Linear types implemented by **Rust** for concurrency.

- **NetBricks operators** consumes packet reference.
NetBricks: Packet Isolation

• Linear types implemented by Rust for concurrency.

• NetBricks operators consumes packet reference.

• API is designed so that safe code can never learn packet buffer address.
NetBricks: Packet Isolation

- Linear types implemented by Rust for concurrency.
- NetBricks operators consumes packet reference.
- API is designed so that safe code can never learn packet buffer address.
- Assuming compiler is sound - packet isolation is guaranteed.
NetBricks Runtime Architecture

Single Process Space

- NF A
- NF B
- NF C
- NF D
- NF X
- NF Y
- NF Z

DPDK Poll for I/O
Scheduler
NICs
NetBricks Runtime Architecture

Single Process Space

DPDK Poll for I/O
Scheduler
NICs

**DPDK**: Fast packet I/O.
NetBricks Runtime Architecture

Single Process Space

NF Chains: Units of scheduling
NetBricks Runtime Architecture

Single Process Space

NF A

NF B

NF C

NF D

NF X

NF Y

NF Z

DPDK Poll for I/O

Scheduler

NICs

Function Call
NetBricks Runtime Architecture

Single Process Space

- Do not preempt NF chain.

- NF A
- NF B
- NF C
- NF D
- NF X
- NF Z

Run to Completion Scheduling

DPDK Poll for I/O
Scheduler

NICs
NetBricks Runtime Architecture

Single Process Space

- Do not preempt NF chain.
- Reduces number of packets in-flight.

DPDK Poll for I/O
Scheduler

NICs
NetBricks Runtime Architecture

Single Process Space

- Do not preempt NF chain.
- Reduces number of packets in-flight.
- Reduces working set size.

NF D
NF C
NF B
NF A

Run to Completion Scheduling

DPDK Poll for I/O
Scheduler

NICs
NetBricks Runtime Architecture

Single Process Space

- Do not preempt NF chain.
- Reduces number of packets in-flight.
- Reduces working set size.
- Preemption points added using queues

DPDK Poll for I/O
Scheduler
NICs
Benefits of Software Isolation

- Provides low overhead *memory* and *packet isolation*. 
Benefits of Software Isolation

- Provides low overhead **memory** and **packet isolation**.
- Improved **consolidation**: multiple NFs can share a core.
Benefits of Software Isolation

- Provides low overhead memory and packet isolation.
- Improved consolidation: multiple NFs can share a core.
- Context switch (~1µs) vs function call to NF (~ few cycles = few ns).
Benefits of Software Isolation

- Provides low overhead memory and packet isolation.
- Improved consolidation: multiple NFs can share a core.
  - Context switch (~1µs) vs function call to NF (~ few cycles = few ns).
- Reduce memory and cache pressure.
Benefits of Software Isolation

- Provides low overhead **memory** and **packet isolation**.
- Improved **consolidation**: multiple NFs can share a core.
  - **Context switch** (~1µs) vs **function call** to NF (~ few cycles = few ns).
- Reduce **memory** and **cache pressure**.
- Zero copy I/O => do not need to copy packets around.
Evaluation Setup

vSwitch

NF1

NF2

VM/Container
Evaluation Setup

VM/Container

NetBricks

vSwitch  NF1  NF2

NF2  NF1
Evaluation Setup

VM/Container

NetBricks

NetBricks Multicore
Evaluation Setup

VM/Container

NetBricks

NetBricks Multicore
NetBricks: More Efficient

Graph showing the processing rate (Mpps) against processing cycles per packet for different configurations:
- NetBricks (3 Core)
- NetBricks (1 Core)
- Bess Container (3 Core)
- OVS Container (3 Core)
- Bess VM (3 Core)
- OVS VM (3 Core)
NetBricks: More Efficient

Processing Rate (Mpps)

3.7x

Processing Cycles Per Packet

NetBricks (3 Core)
NetBricks (1 Core)
Bess Container (3 Core)
OVS Container (3 Core)
Bess VM (3 Core)
OVS VM (3 Core)
NetBricks: More Efficient

![Graph showing processing rate against processing cycles per packet]

- NetBricks (3 Core)
- NetBricks (1 Core)
- Bess Container (3 Core)
- OVS Container (3 Core)
- Bess VM (3 Core)
- OVS VM (3 Core)

Processing Rate (Mpps)

Processing Cycles Per Packet

3.7x

3.53x
Scaling with Chain Length

![Graph showing the processing rate (Mpps) with different chain lengths for NetBricks Multicore, NetBricks, and Container VM. The graph indicates that the processing rate increases with the chain length for all three options.](image-url)
Scaling with Chain Length

![Bar chart showing processing rate (Mpps) vs chain length with NetBricks Multicore, NetBricks, Container, and VM categories. The chart highlights a 1.5x increase in processing rate with chain length.]
Scaling with Chain Length

![Graph showing processing rate with chain length. The graph compares NetBricks Multicore and NetBricks against Container and VM. The processing rate increases with chain length.]

- NetBricks Multicore: Processing rate increases from 1.5x at chain length 1 to 9.3x at chain length 8.
- NetBricks: Processing rate increases from 1.5x at chain length 1 to 9.3x at chain length 8.
- Container: Processing rate increases from 1.5x at chain length 1 to 9.3x at chain length 8.
- VM: Processing rate increases from 1.5x at chain length 1 to 9.3x at chain length 8.
Effect of Increasing Packet Size

![Bar chart showing the effect of increasing packet size on processing rate. The chart compares NetBricks Multicore, NetBricks, and Container VM across different packet sizes (64, 128, 256, 512, 768, 1024, 1200, 1500). The processing rate is measured in Mpps (million packets per second).]
Effect of Increasing Packet Size

Median sized packets

Packet Size

Processing Rate (Mpps)

NetBricks Multicore
NetBricks
Container
VM

64 128 256 512 768 1024 1200 1500
Effect of Increasing Packet Size

Median sized packets

Processing Rate (Mpps)

Packet Size

NetBricks Multicore
NetBricks

Container
VM

6x
Effect of Increasing Packet Size

Median sized packets

Processing Rate (Mpps)

Packet Size

NetBricks Multicore
NetBricks
Container
VM

 Millions of Packets Per Second

6x

≈ 1.42x
NetBricks: Programming Environment
NetBricks Approach

- Write NFs using a **compact set of abstractions** provided by the framework.
NetBricks Approach

• Write NFs using a compact set of abstractions provided by the framework.

• Safe mechanisms whose performance is comparable to native code.
NetBricks Approach

• Write NFs using a compact set of abstractions provided by the framework.
  
• Safe mechanisms whose performance is comparable to native code.
  
• Abstractions implement micro-optimizations to achieve performance.
NetBricks Approach

• Write NFs using a **compact set of abstractions** provided by the framework.
  
  • Safe mechanisms whose **performance** is **comparable** to native code.
  
  • Abstractions implement **micro-optimizations** to achieve performance.

• **User defined functions (UDFs)** provide flexibility.
NetBricks Approach

• Write NFs using a **compact set of abstractions** provided by the framework.

• Safe mechanisms whose **performance** is **comparable** to native code.

• Abstractions implement **micro-optimizations** to achieve performance.

• **User defined functions (UDFs)** provide flexibility.

• Insight: customization is largely orthogonal to performance
NetBricks Approach

• Write NFs using a **compact set of abstractions** provided by the framework.

• Safe mechanisms whose **performance** is **comparable** to native code.

• Abstractions implement **micro-optimizations** to achieve performance.

• **User defined functions (UDFs)** provide flexibility.

• Insight: customization is largely orthogonal to performance.

• Framework can implement **global optimization**.
Abstractions

Packet Processing
- Parse/Deparse
- Transform
- Filter

Control Flow
- Group By
- Shuffle
- Merge

Byte Stream
- Window
- Packetize

State
- Lookup Tables
- LPM Tables
# Abstractions

## Packet Processing
- Parse/Deparse
- Transform
- Filter

## Control Flow
- Group By
- Shuffle
- Merge

## Byte Stream
- Window
- Packetize

## State
- Lookup Tables
- LPM Tables

<table>
<thead>
<tr>
<th>Packet Processing</th>
<th>Control Flow</th>
<th>Byte Stream</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse/Deparse</td>
<td>Group By</td>
<td>Window</td>
<td>Lookup Tables</td>
</tr>
<tr>
<td>Transform</td>
<td>Shuffle</td>
<td>UDF</td>
<td>Consistency</td>
</tr>
<tr>
<td>Filter</td>
<td>Merge</td>
<td>UDF</td>
<td>LPM Tables</td>
</tr>
</tbody>
</table>

- UDF: User-Defined Function
- LPM: longest prefix match

- Consistency
Abstractions and UDFs

... parse::<MacHeader>()
.parse::<IpHeader>()
.parse::<TcpHeader>()
.filter(|pkt| {
    f(pkt.header().src_port())
})
Abstractions and UDFs

... 

`.parse::<MacHeader>()`
`.parse::<IpHeader>()`
`.parse::<TcpHeader>()`
`.filter(| pkt | {
    f(pkt.header().src_port())
})`
Abstractions and UDFs

... parse::<MacHeader>()
parse::<IpHeader>()
parse::<TcpHeader>()

filter(|pkt| {
    f(pkt.header().src_port())
})
Abstractions and UDFs

... parse::<MacHeader>()
    .parse::<IpHeader>()
    .parse::<TcpHeader>()
    .filter(|pkt| {
        f(pkt.header().src_port())
    })
Abstractions and UDFs

...  
parse::<MacHeader>()
parse::<IpHeader>()
parse::<TcpHeader>()

filter(|pkt| {
    f(pkt.header().src_port())
})
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).
- Main contribution: a novel consistent hashing algorithm.
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).

- Main contribution: a novel consistent hashing algorithm.

  - Most of the work in common optimization: batching, scaling cross core.
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).
- Main contribution: a **novel consistent hashing algorithm**.
  - Most of the work in common optimization: batching, scaling cross core.
- NetBricks implementation: **105 lines, 2 hours of time**.
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).
- Main contribution: a novel consistent hashing algorithm.
  - Most of the work in common optimization: batching, scaling cross core.
- NetBricks implementation: **105 lines, 2 hours of time**.
- NetBricks performance (1 core): **9.2 MPPS**
Example NF: Maglev

- **Maglev**: Load balancer from Google (NSDI’16).

- Main contribution: a novel consistent hashing algorithm.

- Most of the work in common optimization: batching, scaling cross core.

- NetBricks implementation: **105 lines, 2 hours of time**.

- NetBricks performance (1 core): **9.2 MPPS**

- Reported: **2.6 MPPS**
Example NF: Evolved Packet Core

- **EPC**: A common NF used in cellular data processing.
Example NF: Evolved Packet Core

- **EPC**: A common NF used in cellular data processing.

- Made by **collaborators** at Berkeley - changes EPC architecture.
Example NF: Evolved Packet Core

- **EPC**: A common NF used in cellular data processing.
- Made by **collaborators** at Berkeley - changes EPC architecture.
- Approximately **2,054** lines of code vs **80,000** for **OpenAirInterface**.
Example NF: Evolved Packet Core

- **EPC**: A common NF used in cellular data processing.
- Made by collaborators at Berkeley - changes EPC architecture.
- Approximately 2,054 lines of code vs 80,000 for OpenAirInterface.
- 10x better performance than OpenAirInterface.
Example NF: Evolved Packet Core

- **EPC**: A common NF used in cellular data processing.
- Made by **collaborators** at Berkeley - changes EPC architecture.
- Approximately 2,054 lines of code vs 80,000 for **OpenAirInterface**.
- 10x better performance than **OpenAirInterface**.
- More than 5x better than commercial **EPCs** based on DPDK.
Upgrading NFs through abstractions
Upgrading NFs through abstractions

Warning: Future work ahead.
Abstractions Enable Upgrades

• Assumed to be the most complex part of the code.
Abstractions Enable Upgrades

• Assumed to be the **most complex part** of the code.

• **Developed** and **provided** by the framework not by NF developer.
Abstractions Enable Upgrades

- Assumed to be the **most complex part** of the code.
- **Developed** and **provided** by the framework not by NF developer.
- Upgrade strategy: Implement **multiple versions** of each abstraction.
Abstractions Enable Upgrades

• Assumed to be the most complex part of the code.

• Developed and provided by the framework not by NF developer.

• Upgrade strategy: Implement multiple versions of each abstraction.

• Each version targets specific hardware feature or software architecture.
Abstractions Enable Upgrades

- Assumed to be the **most complex part** of the code.

- **Developed and provided** by the framework not by NF developer.

- Upgrade strategy: Implement **multiple versions** of each abstraction.
  
  - Each version **targets specific hardware feature** or software architecture.

- Choose which version to use at deployment time.
Abstractions Enable Upgrades

- Assumed to be the most complex part of the code.
- Developed and provided by the framework not by NF developer.
- Upgrade strategy: Implement multiple versions of each abstraction.
  - Each version targets specific hardware feature or software architecture.
- Choose which version to use at deployment time.
  - Choice depends on what is supported, and resource scheduling.
Upgrading Abstractions: Stateless NFs

Lookup Table (Current)
Upgrading Abstractions: Stateless NFs

Lookup Table (Current)

Abstraction backed by local memory
Upgrading Abstractions: Stateless NFs

Lookup Table (Current)

Abstraction backed by local memory

Used by UDFs in other operators
Upgrading Abstractions: Stateless NFs

Lookup Table (Current)

Abstraction backed by local memory

Used by UDFs in other operators

Adopting Stateless Abstraction
Upgrading Abstractions: Stateless NFs

Lookup Table (Current)
- Abstraction backed by local memory
- Used by UDFs in other operators

Adopting Stateless Abstraction
- Abstraction backed by remote KV-store
Upgrading Abstractions: Stateless NFs

**Lookup Table (Current)**
- Abstraction backed by local memory
- Used by UDFs in other operators

**Adopting Stateless Abstraction**
- Abstraction backed by remote KV-store
- UDFs remain unchanged.
Upgrading Abstractions: Stateless NFs

Adopting Stateless Abstraction

- Abstraction backed by remote KV-store
- UDFs remain unchanged.

Adopting Stateless and Caching

- Use consistency requirements to implement local caching.
- UDFs remain unchanged.
Upgrading Abstractions: Stateless NFs

Adopting Stateless Abstraction

Adopting new features requires **no changes** to NF code.

Becomes a **policy decision** made by **deployment**.

Adopting Stateless and Caching

UDFs remain unchanged.
Upgrading Abstractions: Shuffles as RSS

- For many UDFs can implement on NIC.
- Using receive side scaling (RSS).
- RSS can be used when shuffling by TCP 5-tuple
- Masked parts of the IP header.
- Currently implemented.
- Significant performance benefits.

Partition traffic across cores

Shuffle Abstraction

Shuffle

Core 0
Core 1
Core 2
Core 3
Upgrading Abstractions: Shuffles as RSS

UDF dictates how traffic is split.

• For many UDFs can implement on NIC.
• Using receive side scaling (RSS).
• RSS can be used when shuffling by:
  • TCP 5-tuple
  • Masked parts of the IP header.
• Currently implemented.
• Significant performance benefits.
Upgrading Abstractions: Challenges

• Compilations: How to compile UDFs on offload hardware?
Upgrading Abstractions: Challenges

• Compilations: How to compile UDFs on offload hardware?

• Everything is compiled through LLVM. Supports large number of backends.
Upgrading Abstractions: Challenges

• Compilations: How to compile **UDFs** on offload hardware?

• Everything is compiled through LLVM. Supports large number of backends.

• Expectation: Get UDFs in LLVM IR form and retarget as appropriate.
Upgrading Abstractions: Challenges

- Compilations: How to compile UDFs on offload hardware?
  - Everything is compiled through LLVM. Supports large number of backends.
  - Expectation: Get UDFs in LLVM IR form and retarget as appropriate.
- Using Offloads Across NFs: How to share resources or compose, etc.
Upgrading Abstractions: Challenges

- Compilations: How to compile UDFs on offload hardware?
  - Everything is compiled through LLVM. Supports large number of backends.
  - Expectation: Get UDFs in LLVM IR form and retarget as appropriate.
- Using Offloads Across NFs: How to share resources or compose, etc.
  - Example: How to shuffle in chained NFs? Who gets to use an FPGA?
Upgrading Abstractions: Challenges

- Compilations: How to **compile UDFs** on offload hardware?
  - Everything is compiled through LLVM. Supports large number of backends.
  - Expectation: Get UDFs in LLVM IR form and retarget as appropriate.

- Using Offloads Across NFs: How to share resources or compose, etc.
  - Example: How to shuffle in chained NFs? Who gets to use an FPGA?
  - Relying on resource allocation policy to help with these questions.
Conclusion

- NetBricks is a new NF development and execution platform.
Conclusion

• NetBricks is a new NF development and execution platform.

• Addresses three challenges in today’s environments.
Conclusion

• NetBricks is a new NF development and execution platform.

• Addresses three challenges in today’s environments.

  • Providing isolation without overheads.
Conclusion

- NetBricks is a new NF development and execution platform.
- Addresses three challenges in today’s environments.
  - Providing isolation without overheads.
  - Simplifying NF development.
Conclusion

- NetBricks is a new NF development and execution platform.
- Addresses three challenges in today’s environments.
  - Providing isolation without overheads.
  - Simplifying NF development.
  - Enabling NFs to take advantage of hardware and software improvements.
Conclusion

• NetBricks is a new NF development and execution platform.

• Addresses three challenges in today’s environments.
  • Providing isolation without overheads.
  • Simplifying NF development.
  • Enabling NFs to take advantage of hardware and software improvements.

• NetBricks is open sources, available at http://netbricks.io/