Correctness and Performance for Stateful Chained Network Functions

Junaid Khalid$^{W,G}$ and Aditya Akella$^W$

*This work does not have any affiliation with Google*
Network Function Virtualization (NFV)

Hardware NF $\rightarrow$ software NF over commodity server
Network Function Virtualization (NFV)

Hardware NF → software NF over commodity server

*Intrusion detection system (IDS)*
Network Function Virtualization (NFV)

Hardware NF → software NF over commodity server

Intrusion detection system (IDS)

Caching proxy
Network Function Virtualization (NFV)

Hardware NF → software NF over commodity server

- Intrusion detection system (IDS)
- Caching proxy
- Firewall
Network Function Virtualization (NFV)

Hardware NF → software NF over commodity server

- Intrusion detection system (IDS)
- Caching proxy
- Firewall
- WAN optimizer
Network Function Virtualization (NFV)

Hardware NF → software NF over commodity server

- Enables **resource consolidation**
- **Dynamic allocation** of packet processing
- Adding **new functionality**

- **Intrusion detection system (IDS)**
- **Caching proxy**
- **Firewall**
- **WAN optimizer**
Hardware NF → software NF over commodity server

- Enables **resource consolidation**
- **Dynamic allocation** of packet processing
- Adding **new functionality**
- Simplifies **service chaining**
Service Chaining
Service Chaining
Service Chaining

Intrusion detection system (IDS)

WAN optimizer
Service Chaining
Service Chaining
**Service Chaining**

**Chain output equivalence (COE):** for any input the aggregate output of a dynamic set of instances should be equivalent to the output produced by a single instance.
Service Chaining

Chain output equivalence (COE): for any input the aggregate output of a dynamic set of instances should be equivalent to the output produced by a single instance.

Our goal is to provide COE in service chaining **without compromising** performance or correctness.
Service Chaining

Chain output equivalence (COE): for any input the aggregate output of a dynamic set of instances should be equivalent to the output produced by a single instance.

Our goal is to provide COE in service chaining without compromising performance or correctness.

Ensuring COE is challenging: NF chain attributes & Dynamic Actions.
NF Chain Attributes
1. NF statefulness

NF Chain Attributes

• Perform sophisticated *stateful* actions on packets/flows
NF Chain Attributes

• Perform sophisticated *stateful* actions on packets/flows

IDS maintains *cross-flows state* (e.g., per host active conn. count) and *per-flow state* (e.g., TCP conn. state)
NF Chain Attributes

• Perform sophisticated *stateful* actions on packets/flows

IDS maintains *cross-flows state* (e.g., per host active conn. count) and *per-flow state* (e.g., TCP conn. state)
NF Chain Attributes

1. NF statefulness

- Perform sophisticated *stateful* actions on packets/flows
NF Chain Attributes

1. NF statefulness
2. Consistent state updates

- Perform sophisticated *stateful* actions on packets/flows
- Action taken by an NF instance depends on the state updates from other NF instances
NF Chain Attributes

- Perform sophisticated *stateful* actions on packets/flows
- Action taken by an NF instance depends on the state updates from other NF instances

1. NF statefulness
2. Consistent state updates
NF Chain Attributes

1. NF statefulness
2. Consistent state updates
3. Dependency between different NF instances

- Perform sophisticated *stateful* actions on packets/flows
- Action taken by an NF instance depends on the state updates from other NF instances
- Action at the downstream NF may depend on the upstream NFs
NF Chain Attributes

1. NF statefulness
2. Consistent state updates
3. Dependency between different NF instances

- Perform sophisticated *stateful* actions on packets/flows
- Action taken by an NF instance depends on the state updates from other NF instances
- Action at the downstream NF may depend on the upstream NFs
Dynamic Actions

Key requirements
Dynamic Actions

Load balancing/elastic scaling

- Flows are moved from one instance to another to balance load or handle traffic spikes
Dynamic Actions

Load balancing/elastic scaling

• Flows are moved from one instance to another to balance load or handle traffic spikes

Key requirements

• Safe cross-instance state transfer
Dynamic Actions

Load balancing/elastic scaling

- Flows are moved from one instance to another to balance load or handle traffic spikes

Key requirements

- Safe cross-instance state transfer
Dynamic Actions

Load balancing/elastic scaling

- Flows are moved from one instance to another to balance load or handle traffic spikes

Key requirements

- Safe cross-instance state transfer
- Consistent shared state
Dynamic Actions

Key requirements

- Safe cross-instance state transfer
- Consistent shared state
Dynamic Actions

Failure recovery

• When NF fails, all its state disappears. For fault tolerance, that state needs to be recovered

Key requirements

• Safe cross-instance state transfer
• Consistent shared state
• State availability
Dynamic Actions

Key requirements

- Safe cross-instance state transfer
- Consistent shared state
- State availability
Dynamic Actions

Instance slowdown

- Clones may be launched to handle a straggler NF (a slow NF)

Key requirements

- Safe cross-instance state transfer
- Consistent shared state
- State availability
- Duplicate suppression
Dynamic Actions

Instance slowdown
• Clones may be launched to handle a straggler NF (a slow NF)
• Downstream NFs rely on the order at upstream NFs

Key requirements
• Safe cross-instance state transfer
• Consistent shared state
• State availability
• Duplicate suppression
• Chain-wide ordering
Key Requirements for COE
Key Requirements for COE

NF chain attributes

- NF statefulness
- Consistent state updates
- Dependency between different NF instances
Key Requirements for COE

NF chain attributes

- NF statefulness
- Consistent state updates
- Dependency between different NF instances

Dynamic actions

- Elastic scaling
- Failure recovery
- Instance slowdown
Key Requirements for COE

**NF chain attributes**
- NF statefulness
- Consistent state updates
- Dependency between different NF instances

**Dynamic actions**
- Elastic scaling
- Failure recovery
- Instance slowdown

**Key requirements**
- Safe cross-instance state transfer
- Consistent shared state
- State availability
- Duplicate suppression
- Chain-wide ordering
# Existing Solutions

<table>
<thead>
<tr>
<th>Framework</th>
<th>State availability</th>
<th>State transfer</th>
<th>Consistent shared state</th>
<th>Duplicate suppression</th>
<th>Chain-wide ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split/Merge[^NSDI'13]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenNF[^SIGCOMM'14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FTMB [^SIGCOMM'15]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6 [^NSDI'18]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pico Rep.[^SOCC'13]</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StatelessNF[^NSDI'17]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[^NSDI'13]: Split/Merge presented at the Networked Systems and Distributed Computing (NSDI) conference in 2013.

[^SIGCOMM'14]: OpenNF presented at the ACM SIGCOMM conference in 2014.

[^SIGCOMM'15]: FTMB presented at the ACM SIGCOMM conference in 2015.

[^NSDI'18]: S6 presented at the Networked Systems and Distributed Computing (NSDI) conference in 2018.

### Existing Solutions

<table>
<thead>
<tr>
<th>Framework</th>
<th>State availability</th>
<th>State transfer</th>
<th>Consistent shared state</th>
<th>Duplicate suppression</th>
<th>Chain-wide ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split/Merge[NSDI'13]</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenNF[SIGCOMM'14]</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td>✅</td>
<td></td>
</tr>
<tr>
<td>FTMB [SIGCOMM'15]</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S6 [NSDI'18]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✅</td>
</tr>
<tr>
<td>Pico Rep.[SOCC'13]</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>StatelessNF[NSDI'17]</td>
<td>✅</td>
<td>✅</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Incomplete support → restricted functionality
CHC

CHC is a generic NFV framework to support all of these requirements without trading off *correctness* for *performance* or *functionality*
CHC is a generic NFV framework to support all of these requirements without trading off *correctness* for *performance* or *functionality*.
CHC

CHC is a generic NFV framework to support all of these requirements without trading off *correctness* for *performance* or *functionality*.

CHC consists of three main building blocks.
CHC

CHC is a generic NFV framework to support all of these requirements without trading off *correctness* for *performance* or *functionality*.

CHC consist of three main building blocks:
1. State store external to NF
CHC

CHC is a generic NFV framework to support all of these requirements without trading off correctness for performance or functionality.

CHC consist of three main building blocks:
1. State store external to NF
2. NF state-aware state management algorithms
CHC

CHC is a generic NFV framework to support all of these requirements without trading off correctness for performance or functionality.

CHC consists of three main building blocks:
1. State store external to NF
2. NF state-aware state management algorithms
3. Metadata – logical clock and logs
CHC

CHC is a generic NFV framework to support all of these requirements without trading off correctness for performance or functionality.

CHC consist of three main building blocks:
1. State store external to NF
2. NF state-aware state management algorithms
3. Metadata – logical clock and logs
CHC – State Externalization

NF state is stored in an in-memory external state store (similar to statelessNF)
CHC – State Externalization

NF state is stored in an in-memory external state store (similar to statelessNF)
• This ensures *state availability* and simplifies reasoning about *state ownership* and *concurrency control* across instances
CHC – State Externalization

NF state is stored in an in-memory external state store (similar to statelessNF)
• This ensures **state availability** and simplifies reasoning about **state ownership** and **concurrency control** across instances

Naively **externalizing** the state can **degrade** NF performance
CHC

CHC is a generic NFV framework to support all of these requirements without trading off **correctness** for **performance** or **functionality**

CHC consists of three main building blocks
1. State store external to NF
2. NF state-aware state management algorithms
3. Metadata – logical clock and logs
State Management Strategies
State Management Strategies

- State
  - per-flow
  - cross-flow
State Management Strategies

- State
  - per-flow
    - Any
      - Instance-local caching w/ periodic nonblocking flush
  - cross-flow
State Management Strategies

- **State**
  - **per-flow**
    - **Any**
      - Instance-local caching w/ periodic nonblocking flush
  - **cross-flow**
    - Write rarely (read heavy)
      - Instance-local caching w/ callbacks
State Management Strategies

- **State**
  - **per-flow**
    - **Any**
      - Instance-local caching w/ periodic nonblocking flush
  - **cross-flow**
    - Write rarely (read heavy)
    - Write mostly Read rarely
      - Instance-local caching w/ callbacks
State Management Strategies

State

- per-flow
  - Any
    - Instance-local caching w/ periodic nonblocking flush
  - Write rarely (read heavy)
    - Instance-local caching w/ callbacks
- cross-flow
  - Write mostly Read rarely
  - Write/read often
State Management Strategies

State

- per-flow
  - Any
    - Instance-local caching w/ periodic nonblocking flush

- cross-flow
  - Write rarely (read heavy)
    - Instance-local caching w/ callbacks
  - Write mostly
    - Read rarely
  - Write/read often
    - Operation offloading
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment/Decrement a value</td>
<td>Increment or decrement the value stored at key by the given value</td>
</tr>
<tr>
<td>Push/pop a value to/from list</td>
<td>Push or pop the value in/from the list stored at the given key</td>
</tr>
<tr>
<td>Compare and update</td>
<td>Update the value, if the condition is true</td>
</tr>
</tbody>
</table>
An NF instance can offload operations and instruct the datastore to perform them on its behalf.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment/Decrement a value</td>
<td>Increment or decrement the value stored at key by the given value</td>
</tr>
<tr>
<td>Push/pop a value to/from list</td>
<td>Push or pop the value in/from the list stored at the given key</td>
</tr>
<tr>
<td>Compare and update</td>
<td>Update the value, if the condition is true</td>
</tr>
</tbody>
</table>

The datastore serializes operations issued by different instances for the same shared state object and applies them in the background.
An NF instance can offload operations and instruct the datastore to perform them on its behalf.
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf

Without operation offload

<table>
<thead>
<tr>
<th>NF₁</th>
<th>Datastore</th>
<th>NF₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X=0</td>
<td>NF₁</td>
</tr>
</tbody>
</table>
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload:

\[ \text{NF}_1 \rightarrow \text{Datastore}_{X=0} \rightarrow \text{NF}_2 \]

Operations: \( \text{read}(x), \text{lock}(x) \)

With operation offload:
An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload:

- $NF_1$ reads and locks value $X$ from the datastore.
- $NF_2$ reads and locks value $X$ from the datastore.
- $NF_1$ increments the value of $X$.
- $NF_2$ increments the value of $X$.

Diagram:

- $NF_1$ (left) reads $X$, locks $X$, increments $X$, and reads $X$ again.
- $NF_2$ (right) reads $X$, locks $X$, and then reads and locks $X$ again.
- The datastore (center) contains $X=0$ initially and is updated as $X$ is read and locked by both NFs.

Note: The diagram illustrates the flow of operations and state changes in the context of offloading and state maintenance.
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload

\[ \text{NF}_1 \rightarrow \text{Datastore} \rightarrow \text{NF}_2 \]

- \( \text{read}(X), \text{lock}(X) \)
- \( \text{write}(X), \text{unlock}(X) \)
- \( X = 0, \text{aq_lock}(X) \)
- \( X++ \)
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload

\[
\begin{align*}
\text{NF}_1 & \quad \text{Datastore} \quad X=0 \\
& \quad \text{read}(X), \text{lock}(X) \\
& \quad X=0, \text{aq_lock}(X) \\
& \quad \text{write}(X), \text{unlock}(X) \\
& \quad X++
\end{align*}
\]
An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload:

- **NF₁**
  - \(X=0\)
  - \(X=0, aq\_lock(X)\)
  - \(write(X), unlock(X)\)
  - \(X=2\)

- **Datastore**
  - \(X=0\)
  - \(read(X), lock(X)\)
  - \(X=1, aq\_lock(X)\)
  - \(write(X), unlock(X)\)

- **NF₂**
  - \(X=2\)
  - \(read(X), lock(X)\)
  - \(X++\)
An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload:
- NF1
  - X=0, aq_lock(X)
  - X=0, aq_lock(X)
  - write(X), unlock(X)
  - X=2

With operation offload:
- NF1
  - X=0
- NF2
  - X=1, aq_lock(X)
  - X=1, aq_lock(X)
  - write(X), unlock(X)

X++

Datastore
X=0
X=2
NF1
NF2
NF1
 NF2
An NF instance can offload operations and instruct the datastore to perform them on its behalf.

**State Maintenance - Offloading Operation**

**Without operation offload**

- **NF₁:**
  - X = 0
  - read(X), lock(X)
  - X = 0, aq_lock(X)
  - write(X), unlock(X)
  - X = 2

- **Datastore:**
  - X = 0

- **NF₂:**
  - read(X), lock(X)
  - X = 1, aq_lock(X)
  - write(X), unlock(X)
  - X = 2
  - X++

**With operation offload**

- **NF₁:**
  - X = 0
  - increment(X)

- **Datastore:**
  - X = 0

- **NF₂:**
An NF instance can offload operations and instruct the datastore to perform them on its behalf.
An NF instance can offload operations and instruct the datastore to perform them on its behalf.
State Maintenance - Offloading Operation

An NF instance can offload operations and instruct the datastore to perform them on its behalf.

Without operation offload:
- NF1: `X=0`, `write(X), unlock(X)` → `X=2`, `write(X), unlock(X)`
- NF2: `X=0`, `read(X), lock(X)` → `X=1`, `aq_lock(X)`
- Datastore: `X=0`, `X=2`

With operation offload:
- NF1: `X=0`, `read(X), lock(X)` → `X=1`, `aq_lock(X)`
- NF2: `X=1`, `increment (X)`, `X=2` → `X=2`, `increment (X)`
- Datastore: `X=0`, `X=2`
State Management Strategies

State

per-flow

Any

Instance-local caching w/ periodic nonblocking flush

Write rarely (read heavy)

Instance-local caching w/ callbacks

cross-flow

Write mostly Read rarely
State Management Strategies

State

per-flow

- Any
  - Instance-local caching w/ periodic nonblocking flush

cross-flow

- Write rarely (read heavy)
  - Instance-local caching w/ callbacks
- Write mostly Read rarely
  - Non-blocking operation without caching
State Management Strategies

State

- per-flow
  - Any
    - Instance-local caching w/ periodic nonblocking flush

- cross-flow
  - Write rarely (read heavy)
    - Instance-local caching w/ callbacks
  - Write mostly Read rarely
    - Non-blocking operation without caching
  - Write/read often
    - Depends upon traffic split. Cache, if split allows; flush periodically
CHC

CHC is a generic NFV framework to support all of these requirements without trading off *correctness* for *performance* or *functionality*.

CHC consist of three main building blocks:
1. State store external to NF
2. NF state-aware state management algorithms
3. Metadata – logical clock and logs
CHC adds a “root splitter” at the entry of a chain that:
CHC adds a “root splitter” at the entry of a chain that:

• Root splitter attaches a unique logical clock with each packet. Logical clock is used for *duplication suppression, ordering, and traffic replay*
CHC adds a “root splitter” at the entry of a chain that:

- Root splitter attaches a unique logical clock with each packet. Logical clock is used for *duplication suppression, ordering*, and *traffic replay*
- It also logs all the in-transit packets
CHC adds a “root splitter” at the entry of a chain that:

- Root splitter attaches a unique logical clock with each packet. Logical clock is used for duplication suppression, ordering, and traffic replay
- It also logs all the in-transit packets

CHC encodes state object’s ownership information and logical clock associated with state operations as metadata.
CHC – Elastic Scaling

- CHC marks the last packet going to the old instance and first packet going to the new instance

![Diagram showing CHC process]

- Last pkt
- First pkt
- Root splitter
- Old instance
CHC – Elastic Scaling

• CHC marks the last packet going to the old instance and first packet going to the new instance
• Ownership information encoded as metadata of state objects is used to ensure consistent handover of per-flow state
CHC – Elastic Scaling

- CHC marks the last packet going to the old instance and first packet going to the new instance.
- Ownership information encoded as metadata of state objects is used to ensure consistent handover of *per-flow state*.
- *Cross-flow state* does not require any special handling as operation offloading is used to update it.
CHC provides fault tolerance for:

- NF instance
- Root splitter
- Datastore
CHC provides fault tolerance for:

- NF instance
- Root splitter
- Datastore
NF instance failure recovery:

CHC – Fault Tolerance
NF instance failure recovery:

- Failover instance takes over
NF instance failure recovery:

• Failover instance takes over
• Datastore associates the *failover instance ID* with the relevant state
NF instance failure recovery:
• Failover instance takes over
• Datastore associates the *failover instance ID* with the relevant state
• Root *replays* the packet
NF instance failure recovery:

- Failover instance takes over
- Datastore associates the *failover instance ID* with the relevant state
- Root *replays* the packet
NF instance failure recovery:

• Failover instance takes over
• Datastore associates the *failover instance ID* with the relevant state
• Root *replays* the packet
• Metadata is used to *suppress duplicate* state-update and processing
CHC – Straggler Mitigation
CHC – Straggler Mitigation

• Metadata (logical clocks) is used to suppress duplicate state updates at the datastore and duplicate packets at downstream NFs
Metadata (logical clocks) is used to suppress duplicate state updates at the datastore and duplicate packets at downstream NFs
CHC – Straggler Mitigation

- Metadata (logical clocks) is used to suppress duplicate state updates at the datastore and duplicate packets at downstream NFs
Implementation of CHC
Implementation of CHC

• Prototype is implemented in C++
• Leverages Mellanox messaging accelerator for low latency communication
Implementation of CHC

• Prototype is implemented in C++
• Leverages Mellanox messaging accelerator for low latency communication

• We implemented four NFs on top of CHC
  • NAT
  • Trojan detector
  • Portscan detector
  • Load balancer
Evaluation – Performance

Traditional NF with infinite capacity

<table>
<thead>
<tr>
<th>State variable</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port mapping</td>
<td>per-flow</td>
</tr>
<tr>
<td>Total TCP pkt count</td>
<td>cross flow</td>
</tr>
<tr>
<td>Total IP pkt count</td>
<td>cross low</td>
</tr>
</tbody>
</table>
Evaluation – Performance

- Traditional NF with infinite capacity
- Externalized state operations
## Evaluation – Performance

<table>
<thead>
<tr>
<th>State variable</th>
<th>Scope</th>
<th>State Externalization</th>
<th>Caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port mapping</td>
<td>per-flow</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Total TCP pkt count</td>
<td>cross flow</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Total IP pkt count</td>
<td>cross low</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- **Traditional NF with infinite capacity**
- **Externalized state operations**
- **State externalization with caching**
Evaluation – Performance

<table>
<thead>
<tr>
<th>State variable</th>
<th>Scope</th>
<th>State Externalization</th>
<th>Caching</th>
<th>Asynch. + op offload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port mapping</td>
<td>per-flow</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Total TCP pkt count</td>
<td>cross flow</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Total IP pkt count</td>
<td>cross low</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

- Traditional NF with infinite capacity
- Externalized state operations
- State externalization with caching
- State externalization with caching and asynchronous + offloaded updates
Evaluation – Performance

Less than **0.6µs** increase in the median per-NF packet processing latency
Evaluation – Dynamic Actions
Evaluation – Dynamic Actions
Evaluation – Dynamic Actions
Evaluation – Dynamic Actions

During cross instance state sharing

CDF of pkts

Pkt processing time (usec)

OpenNF
CHC
During cross instance state sharing

75\textsuperscript{th}-ile latency of CHC is 20 times lower than OpenNF
Evaluation – Dynamic Actions

CHC

operation offloading
Evaluation – Dynamic Actions

- CHC: operation offloading
- FTMB: checkpointing every 200ms
Evaluation – Dynamic Actions

CHC

operation offloading

FTMB

checkpointing every 200ms

Ensuing Fault tolerance

CDF of pkts

1.0

0.5

0.0

FTMB

CHC

Pkt processing time (usec)

10^0

10^1

10^2

10^3
Evaluation – Dynamic Actions

Ensuing Fault tolerance

75\textsuperscript{th}-\textsuperscript{ile} latency of CHC is \textbf{6 times} lower than FTMB

- CHC: operation offloading
- FTMB: checkpointing every 200ms
Evaluation

CHC operates at line rate with an end-to-end median per packet processing overhead of **11.3us**
Evaluation

• State management performance
• Metadata overhead
• Correctness requirements:
  • State availability
  • Cross instance state transfer
  • Cross instance state sharing
  • Chain wide ordering
  • Duplication suppression
  • Fault tolerance
• CHC supports output equivalence and high performance state management for NFV chains

• It hides the complexity of handling states during dynamic actions (elastic scaling and failure recovery)

• It relies on managing state external to NFs, but couples it with several caching and state update algorithms to ensure low latency