Enabling In-Network Computation in Remote Procedure Calls

Bohan Zhao*, Wenfei Wu**, Wei Xu*

*Tsinghua University, **Peking University

NSDI 2023
Motivation:
In-network computation (INC) is beneficial to system performance but difficult to program.

Contribution:
Make INC easy to use for normal applications with little performance loss.

Metrics:
Reduce lines of code of INC applications by up to 97%.
INC Customizes Stateful Packet Processing
INC is Widely Used in Many Scenarios

- Server Func Offloading
- Line-rate Computation
- Network Stack Simplification
- Big Data Analysis
- Distributed Training
- Network Monitoring
- Distributed Agreement
INC Provides Higher Throughput

- Eliminate incast to reduce traffic
INC Provides Lower Delay

- Reduce the hops of round trip
Challenges of Developing INC Application

Switch

Server1

Server2

Server3

Static switch memory layout

Complex chip-specific language

Tedious network programming
P4 Programming is Complex than Pseudo code

```c
/* P4_14 Program */
action a_get_tmp() {
    subtract(tmp, smac, dmac);
}
action a_com_zero() {
    // do something
}
table get_tmp {
    actions { a_get_tmp; }
}
table com_zero {
    reads {tmp: exact}
    actions { a_com_zero; }
}
control com_smac_dmac {
    apply(get_tmp);
    apply(com_zero);
}
```
A Programming Model is Needed to Wrap INC

- P4 language is network-centric and focus on **communication**
- Users only take care of **computation**
- RPC adapts INC applications better than other models (e.g., MPI)

```c
/* P4_14 Program */
action a_get_tmp() {
    subtract(tmp, smac, dmac);
}
action a_com_zero() {
    void PushPull(double* data, int length) {
        NewGrad request;
        AgtrGrad reply;
        ClientContext context;
        request.mutable_tensor()->mutable_data()->Add(data, data + length);
        Status status = stub_
            ->Update(&context, request, &reply);
        memcpy(data, reply.tensor().data(),
               length * sizeof(double));
        train(data);
    }
```
Challenges in RPC-based INC Programming

- Interface INC functions
- Support concurrent apps
- High-level data types
- Organize messages
- Reliable computation
- Switch memory management
- Reliable transmission
- Flow control
Switch Program is Complex, but INC Behaviors are Similar

- We identify a minimum set of **primitives** to compose INC applications, named reliable INC primitives (**RIPs**) 
- We hope to use the description of INC primitives (**Netfilter**) to replace switch programs

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Args</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map.addTo</td>
<td>stream</td>
<td>map[stream.key] += stream.value</td>
</tr>
<tr>
<td>Map .get</td>
<td>stream</td>
<td>stream.value = map[stream.key]</td>
</tr>
<tr>
<td>Map.clear</td>
<td>empty</td>
<td>map[stream.key] = 0</td>
</tr>
<tr>
<td>Stream.modify</td>
<td>op,para</td>
<td>stream.value = op(stream.value, para)</td>
</tr>
<tr>
<td>CntFwd</td>
<td>key,th,tgt</td>
<td>cnt[key]++; if cnt[key] == th then forward(tgt) else drop</td>
</tr>
</tbody>
</table>
RIPs Reflect Interaction between Data and Switch Memory

Stream

<table>
<thead>
<tr>
<th>Key1</th>
<th>Key2</th>
<th>Key3</th>
<th>Key4</th>
<th>Key5</th>
<th>Key6</th>
<th>Key7</th>
<th>Stream Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value1</td>
<td>Value2</td>
<td>Value3</td>
<td>Value4</td>
<td>Value5</td>
<td>Value6</td>
<td>Value7</td>
<td></td>
</tr>
</tbody>
</table>

- **Stream.modify**
- **Map.addTo**
- **Map.get**
- **Map.clear**
- **CntFwd**

Virtual Map

| +=Value2 | Value4 | 0   | cnt++ |

Switch Register Memory
NetRPC Programming Examples

Protobuf

INC-enabled data types

Indicating NetFilter file name

Netfilter

Quantization factor

RPC

Reliable INC primitives
We implement RIPs on the programmable switch to support multiple jobs concurrently:
Reliable INC Requires Memory-Efficient Idempotence

- INC requires idempotence in addition
  - a. Sockets only guarantee at-least-once packet transmission
  - b. However, repetitive accumulation on the switch causes incorrect result
  - c. Normal path of some INC applications do not involve servers (on-switch reliability)

- We need to detect resent packets with limited switch memory
Reliable INC Requires Fallback to Fit RPC Calls

- INC can fail due to insufficient switch memory, computation overflow, etc.
- But RPC calls should always succeed eventually
- We implement all RIPs on the hosts. When INC fails, the RPC server can complete computation instead
Utilizing Switch Memory Efficiently Guarantees INC Benefits

- Sufficient switch memory makes INC full effect
- We need a management scheme to utilize switch resource efficiently
- We address switch memory in a key-value level by clients

<table>
<thead>
<tr>
<th>Value Stream</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Stream Key</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value1</td>
<td>Value2</td>
<td>Value3</td>
<td>Value4</td>
<td>Value5</td>
<td>Value6</td>
<td>Value7</td>
<td></td>
</tr>
</tbody>
</table>

Pool-based Streaming

<table>
<thead>
<tr>
<th>Value5</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
<th>Value5</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
</tr>
</thead>
</table>
Utilizing Switch Memory Efficiently Guarantees INC Benefits

<table>
<thead>
<tr>
<th>Key-value Stream</th>
<th>Stream Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 1</td>
<td></td>
</tr>
<tr>
<td>Value1 Value2</td>
<td>Value3</td>
</tr>
<tr>
<td>Value4 Value5</td>
<td>Value6</td>
</tr>
<tr>
<td>Value7</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>On-switch Cache</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key1 Key2 Key3</td>
<td>Key4</td>
</tr>
<tr>
<td>Value1 Value2</td>
<td>Value8</td>
</tr>
<tr>
<td>Value4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key-value Stream</th>
<th>Stream Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 2</td>
<td></td>
</tr>
<tr>
<td>Value1 Value2</td>
<td>Value3</td>
</tr>
<tr>
<td>Value4 Value5</td>
<td>Value6</td>
</tr>
<tr>
<td>Value7</td>
<td>2</td>
</tr>
</tbody>
</table>
On-Host Addressing Requires Handling Client Crash

- NetRPC relies on hosts to manage switch memory correctly
- **Memory leak** happens when the client crashes and loses states
- We apply a two-phase timeout to **recycle** valuable switch memory

---

Phase-1 Timeout

Switch

---

Phase-2 Timeout

Server
NetRPC Evaluation

<table>
<thead>
<tr>
<th>Type</th>
<th>Applications and Existing Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncAgtr</td>
<td>Distributed ML training (ATP, SHARP, SwitchML)</td>
</tr>
<tr>
<td>AsyncAgtr</td>
<td>MapReduce (ASK, NetAccel, Cheetah)</td>
</tr>
<tr>
<td>KeyValue</td>
<td>Cache (NetCache, DistCache), Monitoring (ElasticSketch)</td>
</tr>
<tr>
<td>Agreement</td>
<td>Synchronization (P4xos, NetChain, NetLock)</td>
</tr>
</tbody>
</table>

- Can NetRPC simplify INC programming?
- How does the NetRPC system perform?
- Can NetRPC support concurrent application?
- Can NetRPC guarantee reliability?
Reducing User Code Complexity

- NetRPC reduces lines of code of INC applications by up to 97%
Micro-benchmarks of NetRPC

- NetRPC achieves similar performance (≥90%) to baselines even after programming simplification

<table>
<thead>
<tr>
<th>Metrics</th>
<th>NetRPC</th>
<th>Prior Arts</th>
<th>DPDK</th>
</tr>
</thead>
<tbody>
<tr>
<td>SyncAgtr Goodput(Gbps)</td>
<td>50.55</td>
<td>46.44(ATP)</td>
<td>40.11</td>
</tr>
<tr>
<td>AsyncAgtr Goodput(Gbps)</td>
<td>72.31</td>
<td>73.96(ASK)</td>
<td>45.88</td>
</tr>
<tr>
<td>Voting Delay(μs)</td>
<td>20</td>
<td>22(P4xos)</td>
<td>92</td>
</tr>
<tr>
<td>Monitor Delay(ms)</td>
<td>3.52</td>
<td>3.26(ElasticSketch)</td>
<td>4.05</td>
</tr>
</tbody>
</table>
End-to-end Application Performance

- NetRPC achieves even better training throughput than ATP ($\geq 97\%$)
- NetRPC brings 12% higher throughput than P4xos
Support Multiple Concurrent Applications

- NetRPC can support concurrent INC applications with different types and different numbers

<table>
<thead>
<tr>
<th>Metrics</th>
<th>1APP</th>
<th>4APP</th>
<th>4APP×5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync Goodput(Gbps)</td>
<td>50.55</td>
<td>24.88</td>
<td>24.84</td>
</tr>
<tr>
<td>Async Goodput(Gbps)</td>
<td>72.31</td>
<td>36.01</td>
<td>36.6</td>
</tr>
<tr>
<td>Goodput Sum(Gbps)</td>
<td>N/A</td>
<td>60.89</td>
<td>61.44</td>
</tr>
<tr>
<td>KeyValue Delay(ms)</td>
<td>3.52</td>
<td>3.56</td>
<td>3.85</td>
</tr>
<tr>
<td>AgreementDelay( μ s)</td>
<td>20</td>
<td>21</td>
<td>24</td>
</tr>
</tbody>
</table>
NetRPC shows less performance degradation than prior arts with various packet loss rate.
Conclusion

- **NetRPC:**
  The first framework that integrates INC into the familiar RPC programming model

- **Contribution:**
  Make INC development easier and offer similar or better performance boosts than handcrafted systems

- **Future work:**
  Explore scheduling policies and scale NetRPC to more complex topologies
Thanks!