Microsecond-scale Datacenter Computing with RDMA: Characterization, Optimization and Outlooking

Haibo Chen
Institute of Parallel and Distributed Systems (IPADS)
Shanghai Jiao Tong University

https://ipads.se.sjtu.edu.cn/haibo_chen

Work done with Xingda, Rong, et al.
Disclaimers:
I am a system person, not a networking expert 😊

My view:
How to exploit modern networking for systems software?
Emerging online services demand u-second (tail)latency

Low latency requirements

- Audio/Video
- E-commerce
- Online gaming
- Stock exchange
- High-frequency trading
- VR/AR

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Audio/Video</th>
<th>E-commerce</th>
<th>Online gaming</th>
<th>Stock exchange</th>
<th>High-frequency trading</th>
<th>VR/AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50</td>
<td>100</td>
<td>150</td>
<td>200</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

usec level

SELECT c_discount, c_last, ...
FROM customer, warehouse
WHERE w_id = :w_id ...

< 200us

> 20 networking requests

< 10ms
Datacenters: high-throughput, low latency networking

Source: https://developer.nvidia.com/blog/nvidia-grace-hopper-supercip-architecture-in-depth/
Traditional abstraction does not fit fast networking

Due to (relative) slow CPU for processing the networking requests

- The increase of network exceeds that of CPU
- Traditional networking stack (TCP/IP) has high software processing cost

Network increases faster than the CPU

source: https://redian.news/wxnews/78686
Emerging networking provides new abstractions

Examples: RDMA & SmartNIC
- New abstraction whose implementation can be easily offloaded to the hardware

Remote Direct Memory Access (RDMA)
- New abstraction: one-sided RDMA with a remote memory semantic

SmartNIC
- RDMA-capable NIC that further provides customized offloading capabilities
Outline

Characterization: RDMA and SmartNIC

Case study#1: Co-design with distributed systems: LineFS

Case study#2: Co-design with distributed systems: K/V store

Case study#3: Co-design with operating systems: Fast Remote Fork
Remote Direct Memory Access (RDMA)

From systems’ perspective, RDMA provides two primitives:

- Two-sided RDMA: SEND/RECV (like messaging in traditional network)
- One-sided RDMA: offloading primitive for memory READ/WRITE
Empowering systems w/ RDMA: basic approaches

Case study: in-memory key-value store (KVS), e.g., Redis

- Key operation: Get(K) -> V where K, V are stored on a server
- Get(K) requirement: high throughput & low latency
Empowering systems w/ RDMA: two-sided RDMA

Two-sided RDMA to optimize KVS

- Accelerate the network path with faster alternative
- **Pros**: easy to optimize for existing systems

---

**Cons**: server CPU may become the bottleneck, e.g., 150M reqs/sec (NIC) vs. 70M reqs/sec (CPU)
Empowering systems w/ RDMA: one-sided RDMA

Client directly execute the Get with the help of remote memory READ

- **Pros:** NIC can process READ much faster than server CPU
- **Cons:** network operation amplification

![Diagram showing the process of one-sided RDMA](image)
Debate on how to exploit RDMA optimally

Which primitive is better? (even for the simplest KVS get)

- The detailed design choice also changes with different systems

<table>
<thead>
<tr>
<th></th>
<th>One-sided</th>
<th>Two-sided</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lookup A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read A</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC reply</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Performance**

- ✔️
- ❌

**#Round-trips**

- >= 2
- 1

Choose what?
Characterizing RDMA for systems: which primitive to use?

Hard question, depends on several factors
- Performance feature of different RDMA primitives
- How application’s logic can be implemented with different primitives

Our methodology to answer the question
① A performance analysis on RDMA one-sided & two-sided operation
② A systematic characterization of different primitives using representative systems

[1] Deconstructing RDMA-enabled Transaction Processing: Hybrid is Better!, OSDI’18
Primitive performance analysis (w/ the optimal impl.)

**One-sided primitive implementation**
- Native verbs API since one-sided is hardware-offloaded
- Deploy an optimized event loop to hide network latency

**Two-sided primitive implementation**
- Adopt FaSST-RPC [OSDI’16] [1]

---

[1] FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs, OSDI’16
Systematic characterization with distributed transaction

Why distributed transaction?

– An important RDMA application deployed in modern datacenters
– The usage of RDMA in different TX phases simulate usage patterns of the others
RDMA characterization: brief summary

One-sided RDMA is typically faster w/o network amplification

- Network amplification: use more than one one-sided RDMA for a single request

No single primitive wins all the time
Hybrid is better!

In the **same platform**, the **same protocol**, but **w different choices**

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>V</th>
<th>L</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>FaSSTOCC$^{[1]}$</td>
<td>♋</td>
<td>♋</td>
<td>♋</td>
<td>♋</td>
</tr>
<tr>
<td>DrTM+R</td>
<td>[w cache]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FaRM</td>
<td>[w/o cache]</td>
<td></td>
<td>+ ♋</td>
<td>♋</td>
</tr>
<tr>
<td>DrTM+H</td>
<td></td>
<td>+ ♋</td>
<td></td>
<td>♋</td>
</tr>
</tbody>
</table>

[1] FaSST uses a simplified OCC protocol compared to FaRM & DrTM+R.
RDMA: the good and the bad (from systems’ perspective)

**The good**
- Significantly boost the performance of many distributed applications
- Usually by orders of magnitude

**The bad**
- Choosing the right primitive is challenging
- Need to cope with network amplification to fully use RDMA
The network amplification problem of RDMA

One-sided RDMA is the fastest without network amplification

- Yet for (no-so-complex) application (e.g., key-value store), it needs multiple RDMA roundtrips to execute a single request
The central processing unit in RDMA-capable NIC (RNIC) are NIC cores
- ASIC that implements one-sided and two-sided operations

Can we extend RNIC to support customized offloading primitives? SmartNIC!
From one-sided RDMA to SmartNIC, does it help?

SmartNIC: RNIC equip with a programmable SoC (RNIC + SoC)

Back to our initial case study: Get(\(K\)) \(\rightarrow\) \(V\) in key-value storage
- Use programmability of SmartNIC to execute the Get() as a single primitive
From one-sided RDMA to SmartNIC, does it help?

However, our (naïve) SmartNIC-KVS is only 14% of the RDMA-KVS!!

- RDMA-KVS used: DrTM-KV [SOSP’15], workload: YCSB-C (100% Get)
- SmartNIC-KVS: leverage SEND/RECV to offload Get to the NIC SoC

[1] Fast In-memory Transaction Processing using RDMA and HTM, SOSP’15
Need to systematically characterize SmartNIC!
SmartNIC is more complex than we thought

Many SmartNIC architectures exist

We focus on off-path SmartNIC, a widely used SmartNIC architecture
  – Representative example: NVIDIA Bluefield-2 SmartNIC
Existing studies are insufficient and not systematic

Though existing studies exist\(^1\)\(^2\)\(^3\) provide valuable insights

– They mostly focus on the computation power of SmartNIC
– A known takeaway is that: SmartNIC’s SoC cores are wimpier than the host

![Focus of existing study]

### Memory access speed \(^1\) (lower is better)

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoC</td>
<td>4x</td>
<td>4x</td>
<td>N/A</td>
<td>2x</td>
</tr>
<tr>
<td>Host</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
<td>1x</td>
</tr>
</tbody>
</table>

### CPU scores \(^1\) (higher is better)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SoC</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-core Coremark</td>
<td>0.2x</td>
<td>1x</td>
</tr>
<tr>
<td>Single-core Coremark</td>
<td>0.5x</td>
<td>1x</td>
</tr>
<tr>
<td>DPDK hash_perf</td>
<td>0.3x</td>
<td>1x</td>
</tr>
<tr>
<td>DPDK readwrite_lf_perf</td>
<td>0.3x</td>
<td>1x</td>
</tr>
</tbody>
</table>

---

\(^1\) Offloading distributed applications onto smartnics using ipipe. SIGCOMM'19
\(^2\) Performance characteristics of the bluefield-2 smartnic, arXiv
\(^3\) A dbms-centric evaluation of bluefield dpus on fast networks. ADMS'22
Our work: the most comprehensive study on SmartNIC

An important (and basic) component of NIC: communication, is not well explored

- The **communication paths** of SmartNIC are more complex than other NICs

Path #1: Client $\rightarrow$ NIC $\rightarrow$ Host memory

![Diagram of SmartNIC](image)

Traditional NICs (RDMA or non-RDMA)

[1] Characterizing Off-path SmartNIC for Accelerating Distributed Systems, OSDI'23
Our work: the most comprehensive study on SmartNIC

An important (and basic) component of NIC: communication, is not well explored

– The \textit{communication paths} of SmartNIC are more complex than other NICs

\begin{itemize}
  \item \textbf{Path #1}: Client $\rightarrow$ NIC $\rightarrow$ Host memory
\end{itemize}
Our work: the most comprehensive study on SmartNIC

An important (and basic) component of NIC: communication, is not well explored

- The communication paths of SmartNIC are more complex than other NICs

Path #1: Client → NIC → Host memory

Path #2: Client → NIC → SoC memory
Our work: the most comprehensive study on SmartNIC

An important (and basic) component of NIC: communication, is not well explored

- The communication paths of SmartNIC are more complex than other NICs

Path #1: Client → NIC → Host memory

Path #2: Client → NIC → SoC memory

Path #3: SoC memory ←→ host memory
Our work: the most comprehensive study on SmartNIC

An important (and basic) component of NIC: communication, is not well explored

– The communication paths of SmartNIC are more complex than other NICs

Path #1: Client $\rightarrow$ NIC $\rightarrow$ Host memory

Path #2: Client $\rightarrow$ NIC $\rightarrow$ SoC memory

Path #3: SoC memory $\leftarrow$ host memory

Performance characterization on all them!
Takeaway of the characterization

Each communication path of SmartNIC is imperfect

- Inferior performance (or anomalies)
- Underutilization the SmartNIC

Path #1: Client $\rightarrow$ NIC $\rightarrow$ Host memory

1. Inferior performance vs. RNIC
2. SoC is unused

![Diagram showing communication paths and latency comparison between RNIC and SNIC]
Takeaway of the characterization

Each communication path of SmartNIC is imperfect
- Inferior performance (or anomalies)
- Underutilization the SmartNIC

Path #1: Client → NIC → Host memory
1. Faster access but with anomalies
2. Host CPU & memory unused

Path #2: Client → NIC → SoC memory
1. Faster access but with anomalies
2. Host CPU & memory unused
Takeaway of the characterization

Each communication path of SmartNIC is imperfect
- Inferior performance (anomalies) & underutilization the SmartNIC hardware features
- Underutilization the SmartNIC

Path #1: Client → NIC → Host memory
Path #2: Client → NIC → SoC memory
Path #3: SoC memory ←→ host memory

1. RDMA: poor PCIe utilization
2. DMA: poor throughput
3. NIC cores unused
Path #3: Communication between SoC and Host

Several alternatives to implement Path #3

- The simplest (& easiest to use one): RDMA

Yet, RDMA needs to go through RNICs & PCies

- For networking support

Primitives evaluated

Evaluation setup:
ConnectX-6 (RNIC) vs. Bluefield -2 (SmartNIC)
Both NICs use the same NIC cores
Finding. Path #3 has trade-offs

DMA provides a better latency & PCIe utilization
- But its internal engine is slower than RDMA
- Results in lower throughput

→ Path #3 (RDMA) ➔ Path #3 (DMA)

Primitives evaluated

<table>
<thead>
<tr>
<th>Client</th>
<th>READ</th>
<th>WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RDMA is executed by the NIC
DMA is executed by the SoC

RDMA
DMA

Thpt (Mreqs/sec)

READ payload (Bytes)

16B 64B 256B 1KB 4KB 16KB 64KB

0 5 10 15 20 25

DMA RDMA

RDMA is executed by the NIC
DMA is executed by the SoC
Back to our key-value store example

Our characterization explains why naïve KVS on SmartNIC is slow

1. SoC has wimpy cores (known)
2. Path #3 is slow in terms of latency (RDMA) and throughput (DMA)
3. NIC cores are under-utilized on the SmartNIC

*Which are much faster than the SoC*
Guideline for utilizing SmartNIC

Only utilizing a single data path is insufficient for SNIC

- E.g., only use (1): treats SNIC as a vanilla RNIC (Bluefield-2’s NIC mode)
- E.g., only use (2) should take care of the performance anomalies & wimpy SoC power
- E.g., (3) is a killer for the NIC PCIe bandwidth and suffers from high PCIe latency

We have shown that it is always slower than the same standard RNIC

Some can be mitigated while others cannot (e.g., wimpy SoC)

- E.g., (3) is a killer for the NIC PCIe bandwidth and suffers from high PCIe latency

To mitigate it requires completely redesigning the of—path SmartNIC

Need to consider multi-data paths to fully utilize SmartNIC
Outline

Characterization: RDMA and SmartNIC

Case study#1: Co-design with distributed systems: LineFS

Case study#2: Co-design with distributed systems: K/V store

Case study#3: Co-design with operating systems: Fast Remote Fork
Case study: distributed file system

State-of-the-art solution: LineFS@SOSP’21

- Offload file replication from the host to SmartNIC SoC with RDMA
- Drawbacks: (1) over-usage of PCIe bandwidth; (2) weak SoC computation power

(Simplified) Workload: replicate a file from S1 to S2. Note writes are **concurrent**.

Workflow of a single write:

1. **Host sends to SoC (3)**
   - Host
   - SoC
   - NIC
   - PCIe0 (256Gbps)
   - PCIe1 (256Gbps)
   - Network (200Gbps)

2. **SoC sends to the SNIC (2)**
   - SNIC

Peak replication rate = \( \frac{\text{PCIe1}}{2} = 128 \text{Gbps} \)
Which is far **smaller than the network**!
Improved LineFS with our characterization

Opt1: replace RDMA with more bandwidth-friendly DMA

Opt2: utilize host for file replication if SoC is the bottleneck

Other minor optimizations
– Avoid possible SmartNIC anomalies (Please check our paper!)
Outline

Characterization: RDMA and SmartNIC

Case study#1: Co-design with distributed systems: LineFS

Case study#2: Co-design with distributed systems: K/V store

Case study#3: Co-design with operating systems: Fast Remote Fork
Challenge: container startup is slow

e.g., docker run SOME_IMG python foobar.py
- The foobar executes a simple program
- However, container causes **9,000X slower** to the program’s execution (18s)

MITOSIS\[^1\]: fastest container startup method
- Container startup < **5ms** from a clean server
- Start more than **10,0000** containers on 5 machines in **1 second**
- With a **co-design** of remote fork with RDMA

---

\[^1\] No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing, OSDI’23
How RDMA helps? We first characterize container start

Start containers to run the application code involve many steps

1. Download the container image from a registry
2. Containerization: setup cgroup and namespaces
3. Runtime initialization: initialize Python runtime, import libraries (import pytorch)

```
docker run SOME_IMG python foobar.py
```

Network

① Download image

② Containerization

③ Runtime initialization

Where foobar runs:
Naively using RDMA helps (1) but not for (3)

State-of-the-art optimization can partially address the issues

1. Download image: use two-sided RDMA for the download
2. Containerization: use cgroup and namespace pooling to hide its cost [1]
3. Runtime initialization: ?

```
docker run SOME_IMG python foobar.py
```

Network

① Download image

② Containerization

③ Runtime initialization

Where foobar runs:

[1] SOCK: Rapid Task Provisioning with Serverless-Optimized Containers, ATC’18
Idea: reusing initialized state from other containers

Key observation: runtime initialization == initialize container virtual memory (VM)

- We can inherit it from another container if it has initialized
- To efficiently inherit states across machines, we can leverage one-sided RDMA!

```
docker run borrow SOME_IMG python foobar.py
```
Remote fork: a system primitive for inheriting state

Remote fork is a primitive for starting containers

- Generalize fork to provide an easy-to-understand abstraction to the developers
- Start container across machines like forking a container on a local machine

```
docker prepare SOME_IMG
```

```
docker fork SOME_IMG 192.168.12.113
```
MITOSIS: co-designed remote fork with RDMA

Question: how to implement?

− Imitate local fork in a distributed setting!

Two-sided alternative

− Copy the page table & access the physical page with the messages

One-sided alternative

− Copy the page table by reading from the parent’s kernel’s memory with RDMA READ
− Read the physical page with one-sided RDMA

1. Not utilize one-sided RDMA
2. Extra kthreads deployed

Network amplification problem

[1] LITE Kernel RDMA Support for Datacenter Applications, SOSP’17
MITOSIS: co-designed remote fork with RDMA

A hybrid approach for reducing network amplification of one-sided RDMA

- Use two-sided RDMA to copy the page table of the parent process
- Use one-sided RDMA to read the pages with the help of the page table

1. Mark as copy-on-write
2. Two-sided
3. One-sided

3. Create a container w/ the read page table
MITOSIS: co-designed remote fork with RDMA

44—80% faster than basic C/R[1] not co-designed with RDMA
- The C/R implementation has also been optimized with RDMA

---

[1] The state-of-the-art impl w/o RDMA
Killer application of MITOSIS: serverless computing

A new paradigm on building cloud applications for elasticity

- The platform dynamically start new containers for handling user requests

16—73% latency improvements to Coldstart (FaasNET) w/o image pull

Functions sampled from ServerlessBench@SOCC'21, FunctionBench@SoCC'19 and SeBS@SoCC'21.
Outline

Characterization: RDMA and SmartNIC

Case study#1: Co-design with distributed systems: LineFS

Case study#2: Co-design with distributed systems: K/V store

Case study#3: Co-design with operating systems: Fast Remote Fork
Co-design KVS w/ RDMA: two-sided primitive

Straightforward approach:
- Replacing the communications between the client & server with two-sided RDMA
- E.g., messaging or RPC

Network accelerated with RDMA
Problem: server CPU Is becoming the bottleneck

Trend: increasing CPU-NIC gap
- NIC: 150M messages/sec
- CPU: 70M messages/sec (w/o KVS logic)

[1] StRoM: Smart Remote Memory (EuroSys '20)
Co-design with RDMA: one-sided primitive

RNIC directly reads/writes memory
- Offload index traversal to NIC
- Totally bypass server CPU
Reduce RDMA network amplification w/ learned cache

Observation: learned index\textsuperscript{[1]} — using ML model as index structure

- Index with very small memory footprint that can be cached at the clients

\textsuperscript{[1]} The case for learned index structures. SIGMOD’18
Reduce RDMA network amplification with learned cache
Reduce RDMA network amplification w/ learned cache
Reduce RDMA network amplification w/ learned cache

#1 Constant $O(1)$ lookup time!

#2 Small memory footprint

Get($K_1$) = $V_1$
XStore: a KVS with learned cache + one-sided RDMA

Execution flow of Get(

Client

Server

READs

Index

Values

Client

Server

O(log n) vs. O(1)

Two-sided KVS is bottlenecked by the CPU

One-sided KVS is bottlenecked amplification

[1] Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache
Outlooking
Performance vs. programmability: call for new abstraction

An inherent trade-off in programmability and performance

- NIC core: the fastest but is not programmable
- SoC: the easiest to program but is even slower than the host
- Approaches between NIC core & SoC, e.g., FPGA & P4

Call for good abstraction to embrace both performance and programmability
How to automatically synthesize optimizations?

Developers needs to be “smart” as well to exploit SmartNIC

We used application-specific approaches for optimization
– With domain-specific knowledge
– Long development cycle

Optimization method (program) fails to generalize to other applications
– e.g., learned cache only works for key-value store (or indexing)

Calls for systematic and automatic approach for synthesize optimization
Summary

Characterizing the performance of RDMA & SmartNIC
- RDMA: DrTM+H [OSDI’18]
- SmartNIC: SmartNIC-study [OSDI’23]

Co-designed systems with RDMA & SmartNIC for better performance
- Operating system: MITOSIS [OSDI’23]
- Applications: XStore [OSDI’20]

Outlooking: good abstraction and automatic synthesizing optimizations
Thanks & QA