RMI: High Performance User-level Multi-threading on SPDK/DPDK

Munenori MAEDA†, Shinji KOBAYASHI†, Jun KATO†, Mitsuru SATO†, Hiroki OHTSUJI‡

† Fujitsu Limited
‡ Fujitsu Laboratories Ltd.
RMI Development Team

Munenori MAEDA
- Fujitsu Limited
- TL, Engineer, Ph.D.

Shinji KOBAYASHI
- Fujitsu Limited
- Engineer

Jun KATO
- Fujitsu Limited
- Engineer

Mitsuru SATO
- Fujitsu Limited
- Senior Manager, Ph.D.

Hiroki OHTSUJI (Speaker)
- Fujitsu Laboratories Ltd.
- Researcher, Ph.D.
Agenda

- Background
  - Why Thread?
- RMI Architecture
  - RMI Design and Feature
  - ARCHITECTURE
- RMI Threading Environment
  - Framework Overview
  - Multicore Optimizations
  - Benchmark Result

- RMI Internode Communication
  - Design Overview
  - Multicore Optimizations
  - Performance Evaluation

- Summary
Background
Background

Enterprise AFA Requirements
- Capacity
- Performance
- Reliability and Availability
- Scalability
- Rich Storage Services
  - Deduplication
  - Erasure Coding
  - Cloud native functions
  - ... and so on

HW Technology Trends
- Multicores with Dual or more sockets
- High Speed Interconnect with RDMA (RoCE)
- Low latency Storage Device (NVMe, 3D XPoint)
- SPDK Framework
  - Kernel-bypass Architecture to exploit HW Performance of the latest devices
  - Potential to be de facto

Our research has been started since 2016
Background

What’s the matter of our development? In early evaluation we got

- Development Expense and Speed
  - Developers are unfamiliar with asynchronous programming
  - Debugging takes Long time

- Effective Performance and Scalability
  - Multicore parallelism sometimes degrades performance
  - No load balancer to level CPU utilizations
  - Internode Communication impacts I/O latency and Performance

Missing pieces of SPDK

- Programmability
- RDMA based Internode Communication
What we need for Programming Framework

- Better programmability for Industrial Applications
  - Compact
  - Comprehensive
  - Reliable

- Integrity with SPDK
  - SPDK so far provides basic Storage Services and HW Drivers, including NVMe driver
  - Use them “as is” to shorten a development period
Multi-threading Framework

- **Thread vs. Event**
  - Long time debates in Academic and Industrial communities
  - Both have the Pros and Cons
  - Yes, SPDK/DPDK is in “Event” side

- **Why Thread?**
  - Developers are quite familiar with Blocking API
    - Traditional POSIX programming: file, socket

- **Why Thread? (Contd.)**
  - Multi-threading has great success in some domains:
    - Apache WEB server and WEB services with stateful sessions

- **Suited to AFA Storage Services such that**
  - Complicated and large code
  - Many states to handle HW or SW failures
Internode Communication

- Various Comm. Purpose
  - Delegate I/O request to LUN Node
  - Update Metadata
  - Maintain Data Redundancy
  - Check Sanity/Soundness for HA

- Data Redundancy constrains Scalability
  - Mix. use of Short Messages and Long RDMA transfers (8KB or More)
RMI Architecture
RMI Design and Feature

- **Our Framework: RMI**
  - An abbreviation of Rocket Messaging Infrastructure

- **Supports Multi-threading**
  - Lightweight user-level threads
  - Various synchronizing primitives, similar to Java and Go

- **Enables SPDK compliant**
  - No changes for SPDK NVMe driver and 3rd party driver
  - SPDK event-driven programming is still effective

- **Provides Dynamic Load Balancing**
  - Level multicore utilization to exploit full performance
## ARCHITECTURE

### Storage Protocols
- iSCSI Target
- NVMe-oF Target
- SCSI
- NVMe

### Storage Services
- **RMI Threading Environment**
  - Block Device Abstraction (BDEV)
    - Vendor’s Impl.
  - RMI BDEV Bridge API
  - Thread Pool / Load Balancer
  - Synchronizing Primitives

- **Synchronous Multi-thread Support**

### Drivers
- NVMe Devices
  - NVMe-oF Initiator
  - NVMe PCIe Devices
- Intel QuickData Technology (I/OAT) Driver
- Internode Device
  - RDMA (Verbs) Driver
- Accelerator
  - Intel QuickAssist Technology Driver

### Legend
- RMI
- SPDK
- 3rd Party

Copyright 2019 FUJITSU LIMITED
This Talk ...

- Two main topics, we present, are:
  - RMI Threading Environment
  - RMI Internode Communication

- Meanwhile, the following topics are omitted or only outlined:
  - RMI BDEV Bridge API
  - RMI Load Balancing Algorithm
  - RMI Synchronizing Primitives
  - RMI QuickAssist Driver Glue
RMI Threading Environment

Based on Lthread of DPDK 16.11
- User-level: N OS native threads handle M RMI threads
- "Cooperative", or Non-preemptive: running threads voluntarily yield control

Synchronizing Primitives
- i-structure (the same as Haskell’s ivar, Java’s Future)
- q-structure (the same as Java’s BlockingQueue, Go’s channel)
- RCU, Semaphore, and so on.

Thread Pool/Load Balancer
- Enables fast and efficient allocation of Threads
- Provides a “matching place” among idle threads and tasks
Design Considerations

- Practical Limitation of Memory Footprint
  - How many available threads?
  - It depends on the gross stacks of threads

- Locality-aware Task Scheduling
  - Existing task scheduling methods are:
    1. Tasks activate Idle Threads
    2. Active Threads pull Waiting Tasks
  - Common Problems
    - Locality Unawareness
    - Unnecessary Context Switches
RMI Threading Environment: Thread Pool

**RMI Approach**

- **Fixed Memory Footprint 2GB**
  - Gross stack is the product of 16K threads x 128KB fixed stack for each

- **Hybrid Task Scheduling**
  - Hybrid of existing (1) and (2) methods
    - Idle Threads and Waiting Tasks are both *non-empty* in some condition
    - Lazy binding of Task and Thread
  - Locality-aware
  - Minimized Context Switching
static int number_stream(void) {
    ...
    // Task declaration
    struct rmi_exec_task gen_task;

    // Content of Task is defined
    rmi_exec_task_init(&gen_task);
    gen_task.fn = gen;
    gen_task.fn_arg = &ga;

    // Submit the Task into Thread pool
    rmi_task_submit_thrpool(&gen_task);
    ...
}

static int gen(void *arg) {
    struct gen_arg args = *(struct gen_arg*)arg;

    for (uint64_t i=2; i < args.max; ++i) {
        // use of RMI synchronizing primitive:
        // write integer to bounded-buffer
        rmi_bbstr_u64_write(args.out, i);
    }

    rmi_bbstr_u64_write(args.out, 0UL);
    return 0;
}

Computational Entity defined as Task
RMI Threading Environment: Load Balancing

Design Considerations

Possible Migration Target are:

- Active Running Threads, or
- Runnable Tasks in the Thread Pool

Active Running Threads

- [Pros] LB may directly level core utilizations
- [Cons] C programs and libraries are constrained for the use of Thread Local Storage, TLS
- Usually supported by program language like Go

Runnable Tasks ← RMI only supports

- [Pros] No limitations for C programs including OSS, because TLS is not yet accessed
- [Cons] Behind the task rearrangement, LB may gradually level core utilizations

Native threads may access TLS safely, assisted by OS. Cooperative threads, however, have no OS assist.
RMI and SPDK Integration

- RMI thread scheduler takes over an SPDK reactor, running on each core
- SPDK runs under the RMI scheduler context

**RMI Main Loop**

- Poll HW and SW Queues
- Manage Timers
- SPDK reactor single iteration
- Switch to a thread context

No change, just as-is

- Poller and Timer Poller
- Prioritized Event run
- Event run
Multicore Performance Issue

- SW Queues in RMI Main Loop are Frequently Accessed
- SW Queues contain RMI hub infra., designed as Concurrent Queue:
  - RMI task queue
  - Ready queue
  - SPDK event queue
- Concurrent Queue **may impact** Multicore Performance
  - Mutex are NEVER used in RMI Main Loop in Principle
  - Most Non-blocking Concurrent Queues use HW Locks (atomic instructions) instead
  - HW locks restrict Multicore Scalability
**Preliminary Concurrent Queue Evaluation**

- **Round-trip Test for Concurrent Queue**
  - Single Queue per Core, as “Multi Produces, Single Consumer”
  - Measure Elapsed time from (1) to (4)
  - All-to-all

  ![Diagram](image)

  (1) element push
  (2) element pop
  (3) element push
  (4) element return by pop

- **Comparison**
  - `rte_ring` of DPDK 16.11
  - Vyukov’s queue in Lthread of DPDK 16.11
Preliminary Result

**Result / rte_ring**

<table>
<thead>
<tr>
<th># of cores</th>
<th>round trip time (us)</th>
<th>Single socket: all cores in a single socket</th>
<th>Dual sockets: divide cores in half</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.6369</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.3266</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8.40633</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>11.50558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>22.94589</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16+16</td>
<td>68.56327</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8+8</td>
<td>8+8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Xeon E5-2697 v4 (Broadwell) 2.3GHz 2sockets**
  - Long Latency
    32 core's RTT is 68.5us.
  - Poor Scalability
    RTT is 48x when # of cores is 16x.
Preliminary Result (Contd.)

- **Result / Vyukov’s queue**

  - **Xeon E5-2697 v4 (Broadwell) 2.3GHz 2sockets**

  - **Vyukov’s queue of Lthread**

  - **Single socket: all cores**
    - Acceptable Latency?
      - 32 core’s RTT is 10us.
    - Good Scalability
      - RTT is 18x when # of cores is 16x.

  - **Dual sockets: divide cores in half**
    - 8+8
    - 4+4
    - 16+16

  - **6.6x faster than rte_ring**

  - **Copyright 2019 FUJITSU LIMITED**

  - **23**
RMI Multicore Optimizations

Observations

- HW lock by inter-socket cores is quite heavy
  - High Performance needs to reduce HW locks
- Improving latency for concurrent queue is still necessary
  - Vyukov’s queue is faster, but its latency is relatively high (>5us each way)

RMI Approach

- Optimizations to reduce HW Locks, focusing on some dedicated use:
  - RMI task queue for Load Balancing
  - Lockless i-structure for HW completions
  - RMI Internode Communication (presented later)
RMI task queue

- Consists of $N$ fast SPSC Queues with No HW Lock and slow MPSC queue
- Enables fast push/pop unless SPSC queues are overflowed
- Preserves FIFO order even overflowed

N SPSC queues, similar to SPSC rte_ring

MPSC rte_ring

Push if SPSC queue is overflowed
RTT Benchmark Result

**Result / RMI task queue**

- **Xeon E5-2697 v4 (Broadwell) 2.3GHz 2sockets**
  - Low Latency
    - 32 core’s RTT is 4.8us.
  - Super Scalability
    - RTT is 12x when # of cores is 16x.

---

**# of cores**

- **Single socket:** all cores in a single socket
- **Dual sockets:** divide cores in half

---

**Round trip time (us)**

- **rmi_task_queue**
- **Vyukov’s queue**
Prime Benchmark

Description
- “Prime” sifts N natural numbers, N=10M, with 10K filter threads of primes (2,3,5,...)
- Filter threads are connected by channels
- # of actual threads is less than 16K because of RMI Thread Pool capacity limitation

Compare with a native Go program
- Both use compatible synchronization primitives
- Go program creates threads directly, while RMI program submits tasks in Thread Pool
Prime Benchmark Result

- Good Performance, superior to Go
- Need further analysis for Scalability Decrease

SUPERMICRO 2028U-TN24R4T / Xeon E5-2697 v4 2.3GHz 36 Cores / 512GB
RMI Internode Communication
Verbs API is commonly used in RDMA over Converged Ethernet (RoCE).

We adopt RoCE for low latency and low CPU usage.

Communication primitives for RMI

Using native Verbs API?

- Verbs API provides asynchronous posting and polling style primitives
  - E.g. `ibv_post_send()` and `ibv_poll_cq()`
  - Polling in every RMI threads is complicated and inefficient

RMI Approach

- Provides simple blocking style primitives implemented using Verbs APIs
  - E.g. `mmi_send()`, `mmi_recv()`, `mmi_read_rdma()`, `mmi_write_rdma()`
Implementation of Blocking Primitives

- **Send/RDMA Read/RDMA Write**
  - i-structure is used to synchronize with the I/O completion.

**RMI Main Loop**
- Poll HW and SW Queues
- Manage
- SPDK reactor single iteration
- Switch to a thread context

**RMI Thread**
- 1. Post Send
- 2. Read i-structure
- 3. Poll Send Completion
- 4. Write i-structure
- 5. Resume execution
Receive

- q-structure is used for synchronization as multiple messages may arrive.
Spin Locks in the communication library impact performance

- Performance degrades when many cores access a Queue Pair (QP)
  - QP is the communication endpoint in Verbs API. One QP is needed for each communication target
Provide a dedicated QP for each Destination Node/RoCE Port/Core.

- Each core communicates with the corresponding core on the destination node.
- The illustration below assumes single RoCE Port for simplicity.

---

**Node 1**

- Core 0
- Core 1
- Core N

- QP

---

**Node 2**

- Core 0
- Core 1
- Core N

- QP

---

**Node 3**

- Core 0
- Core 1
- Core N

- QP
Dedicated QP for Each Core (Contd.)

**Pros**
- No simultaneous access for QPs.
  - Spin lock costs are greatly reduced.
  - Specifying an appropriate thread model through Mellanox’s experimental Verbs API further improves CPU usage wasted for unnecessary HW locks.

**Cons**
- Many QPs are needed.
  - Modern NICs supports more than ~100K QPs.
  - Number of QPs needed = Max. # of destination nodes × # of RoCE ports × # of cores
    = 15 × 2 × 40 = 1200

We have enough QPs for our case.
Performance Evaluation Environment

Software

- Latency benchmark using pairs of RMI threads exchanging a message or performing RDMA operations in Ping-Pong style.

Hardware

- Processors: 2 Intel Xeon Gold 6148 (Skylake)
  - 38 cores out of 40 cores are used for experiments.
  - HyperThreading is disabled.
- NIC: Mellanox ConnectX-4 EN
  - Dual port 40GbE.
- Network: 40GbE switch
Performance Evaluation Results

- Latency of Dedicated QP is \(~1/10\) of that of Shared QP when many threads are communicating.
- Optimization of acquiring a lock before polling is adopted for Shared QP.

### Latency of RDMA 512B

- Dedicated QP RDMA Read
- Dedicated QP RDMA Write
- Shared QP RDMA Read
- Shared QP RDMA Write
- Minimum 2 * 40GbE

### Latency of Message Send/Receive

- Dedicated QP
- Shared QP
Performance Evaluation Results (Contd.)

- CPU Usage of `pthread_spin_lock()` is reduced if Exp. Verbs is used.
- We use `IBV_EXP_THREAD_UNSAFE` in `ibv_exp_create_res_domain()`.

**pthread_spin_lock CPU Usage (RDMA 512B)**

- **Read Without Exp. Verbs**: The CPU usage is higher without Exp. Verbs.
- **Read With Exp. Verbs**: The CPU usage is reduced with Exp. Verbs.
- **Write Without Exp. Verbs**: The CPU usage is higher without Exp. Verbs.
- **Write With Exp. Verbs**: The CPU usage is reduced with Exp. Verbs.

**pthread_spin_lock CPU Usage (Message)**

- **Without Exp. Verbs**: The CPU usage is higher without Exp. Verbs.
- **With Exp. Verbs**: The CPU usage is reduced with Exp. Verbs.
Summary

- Designed and Implemented RMI on SPDK/DPDK for Enterprise AFA
  - Better programmability
    - Light-weight user-level threads
    - Rich synchronizing primitives
    - Blocking style communication primitives
  - SPDK compliant
    - Various libraries and HW drivers can be used “as is.”
  - High Performance
    - Load balancing with fast concurrent queue
    - Lockless i-structure
    - Low latency RDMA
Questions?

- E-mail: flab-spdk-dev@ml.labs.fujitsu.com
Fujitsu
shaping tomorrow with you
Bdev Extension

**SPDK design**
- Bdev fn_table may define:
  - What to handle
    - Submit_request
    - Get_io_channel
    - Destruct
  - How to dispatch
    - Function call
    - Event call
- spdk_bdev_io_submit/_complete
  - Invoke Bdev fn_table, like as Vtable

**RMI design**
- Bdev fn_table Extension:
  - Adds RMI’s dispatch manner
    - Task submit
- rmi_bdev_io_submit/_complete
  - Determine whether transferring direction is from SPDK to RMI, or vice verse
  - Select dispatch mechanism
    - [RMI→SPDK] event call
    - [SPDK→RMI] task submit
    - [*→*] Function call
  - Invoke extended Bdev fn_table
Dedicated asynchronous primitives may be called anytime, anywhere to transfer controls between layers.
The HW lock in i-structure can be safely removed if posting Send is performed **AFTER** the RMI Thread is blocked.
Lockless i-structure (Contd.)

- **Generic i-structure**
  - Post Send before reading i-structure.
  - HW lock in i-structure is mandatory as another core may write to the i-structure.

  ```c
  ...  
  post_send(...);
  int ret = rmi_istr_int_read(&istr);
  // Blocks until the Send is completed.
  ```

- **Lockless i-structure**
  - Post Send in the pending function which is executed after the RMI thread is blocked by reading i-structure.
  - HW lock in i-structure is not necessary.

  ```c
  static void pending_send(rmi_istr_int *istr, void *arg) {
    // Executed in the scheduler context just after the RMI
    // Thread is blocked.
    post_send(...);
  }
  ```

  ```c
  ... 
  int ret = rmi_istr_lockless_int_read(&istr,
                                    pending_send, &args);
  // Blocks until the Send is completed.
  ```
Latency of RDMA 8KB is limited by the 40GbE throughput.
CPU Usage of `pthread_spin_lock()` is reduced if Exp. Verbs is used.

- We use `IBV_EXP_THREAD_UNSAFE` in `ibv_exp_create_res_domain()`.
Performance Evaluation Results (Contd.)

- **Shared QP without Lock Optimization has severe Performance Degradation.**

![Latency of RDMA 512B](image1)

![Latency of Message Send/Receive](image2)
Shared QP without Lock Optimization has severe Performance Degradation.