FUJITSU

shaping tomorrow with you

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.

Copyright 2019 FUJITSU LIMITED

RMI Development Team

FUJITSU

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.

Copyright 2019 FUJITSU LIMITED

Agenda

FUJITSU

- 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

Copyright 2019 FUJITSU LIMITED

Background

Copyright 2019 FUJITSU LIMITED

Background

FUJITSU

- 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

Copyright 2019 FUJITSU LIMITED

Background

FUJITSU

- 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

Copyright 2019 FUJITSU LIMITED

What we need for Programming Framework

FUJITSU

- 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

Copyright 2019 FUJITSU LIMITED

Multi-threading Framework

FUJITSU

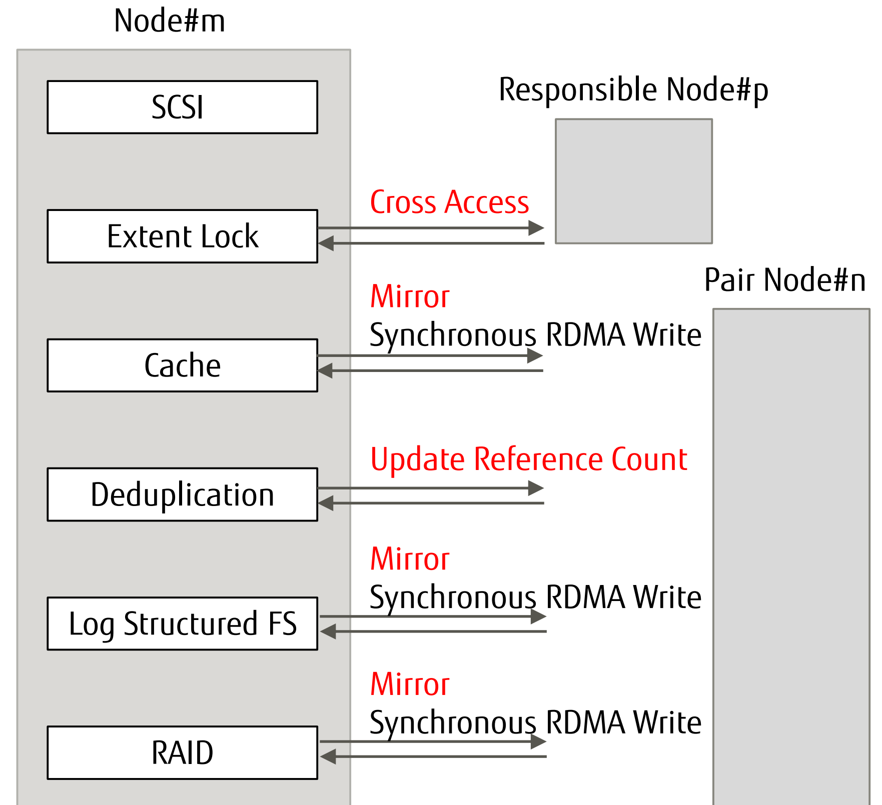
- Thread vs. Event
  - Long time debates in Academic and Industrial communities
  - Both have the Pros and Cons
  - Yes, SPDK/DPDK is in "Event" side
- Why Thread?
  - Developers are quite familiar with Blocking API
    - Traditional POSIX programming: file, socket
- Why Thread? (Contd.)
  - Multi-threading 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

Copyright 2019 FUJITSU LIMITED

Internode Communication

FUJITSU

- 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)



Copyright 2019 FUJITSU LIMITED

RMI Architecture

FUJITSU

Copyright 2019 FUJITSU LIMITED

RMI Design and Feature

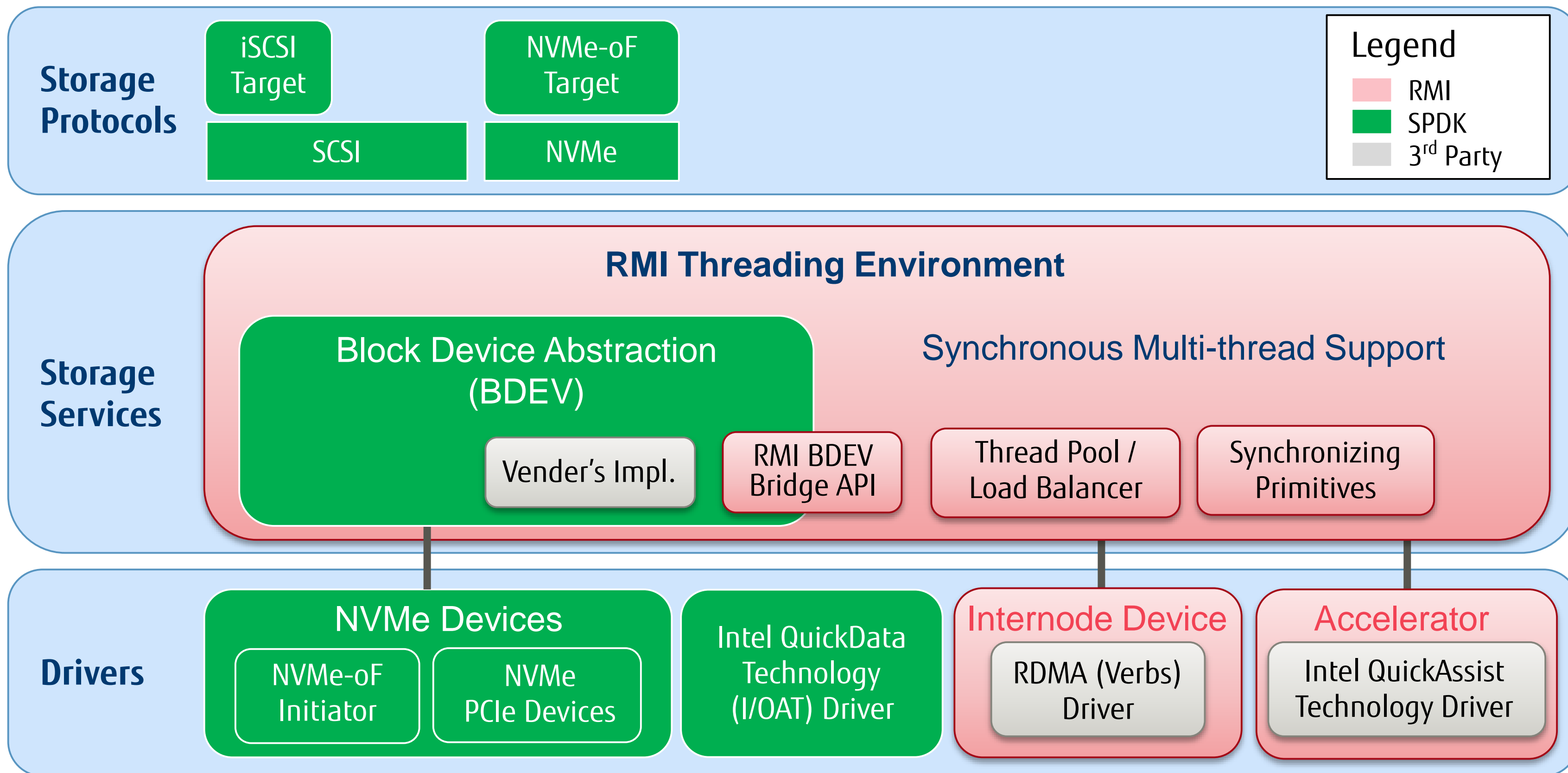
FUJITSU

- 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

Copyright 2019 FUJITSU LIMITED

ARCHITECTURE

FUJITSU



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

Copyright 2019 FUJITSU LIMITED

RMI Threading Environment

Copyright 2019 FUJITSU LIMITED

RMI Threading Environment

FUJITSU

- 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

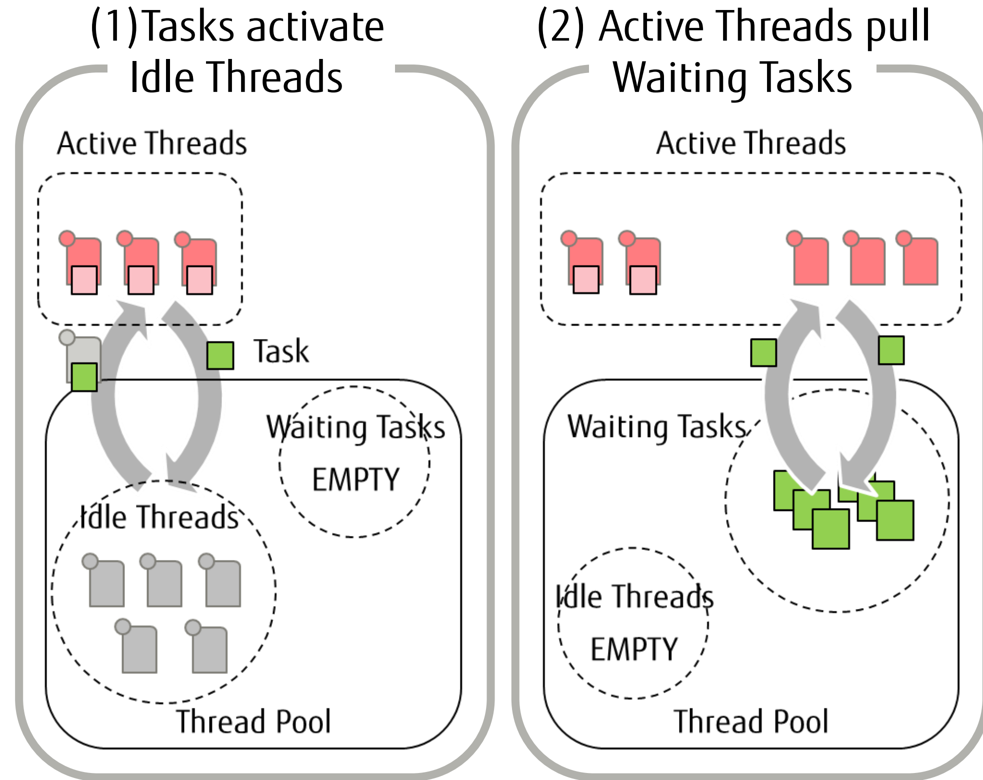
Copyright 2019 FUJITSU LIMITED

RMI Threading Environment: Thread Pool

FUJITSU

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



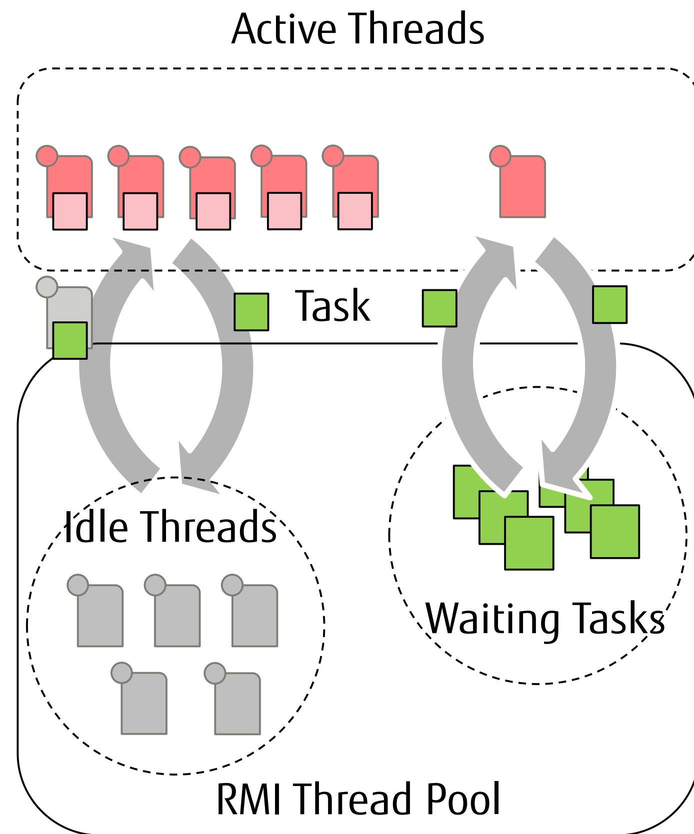
Copyright 2019 FUJITSU LIMITED

RMI Threading Environment: Thread Pool

FUJITSU

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



Copyright 2019 FUJITSU LIMITED

RMI Threading Environment: Example

FUJITSU

```
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

Copyright 2019 FUJITSU LIMITED

RMI Threading Environment: Load Balancing

FUJITSU

- 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

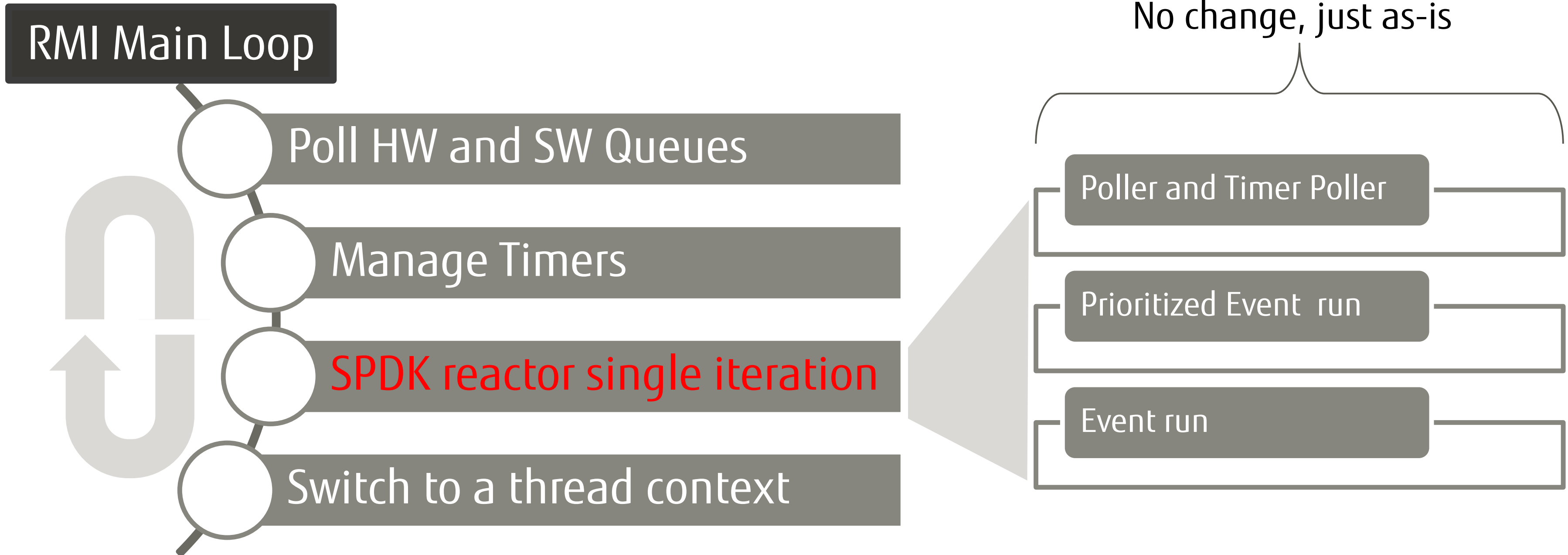
Native threads may access TLS safely, assisted by OS.
Cooperative threads, however, have no OS assist.

Copyright 2019 FUJITSU LIMITED

RMI and SPDK Integration

FUJITSU

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



Copyright 2019 FUJITSU LIMITED

Multicore Performance Issue

FUJITSU

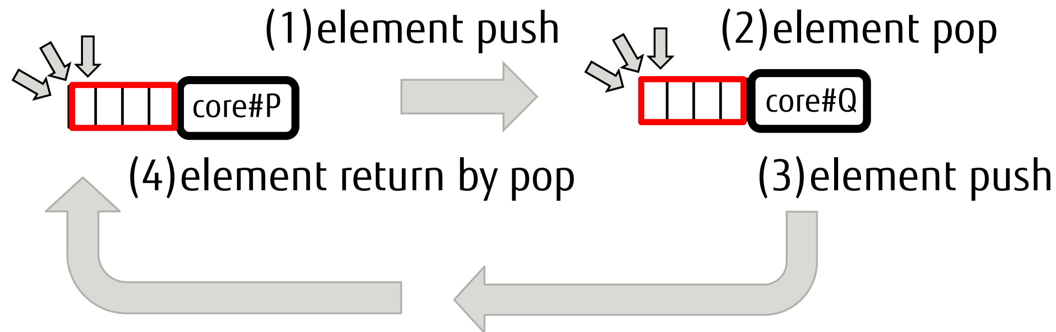
- SW Queues in RMI Main Loop are Frequently Accessed
- SW Queues contain RMI hub infra., designed as Concurrent Queue:
  - RMI task queue
  - Ready queue
  - SPDK event queue
- Concurrent Queue **may impact** Multicore Performance
  - Mutex are NEVER used in RMI Main Loop in Principle
  - Most Non-blocking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

Copyright 2019 FUJITSU LIMITED

Preliminary Concurrent Queue Evaluation

FUJITSU

- 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

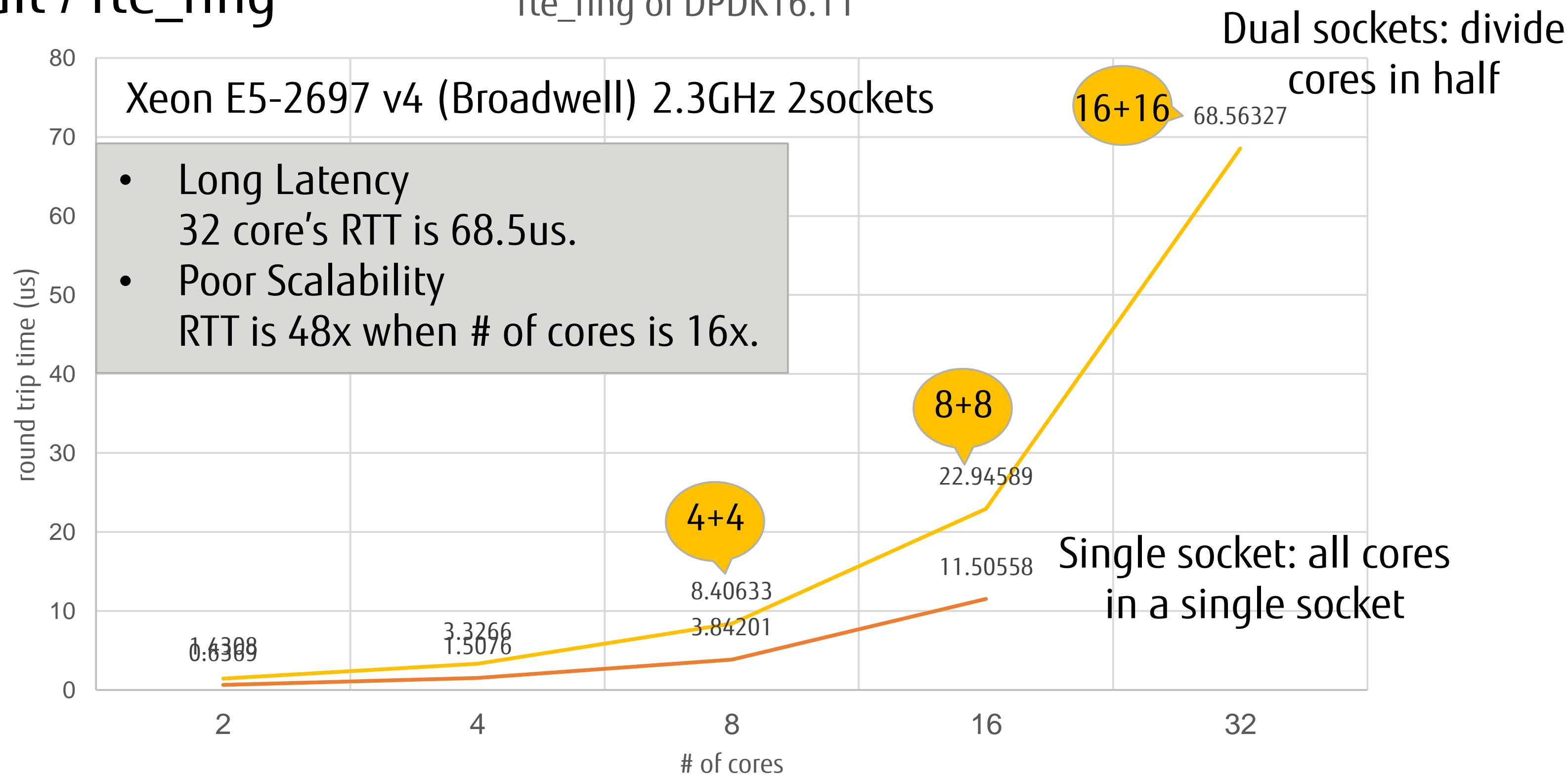


- Comparison
  - rte_ring of DPDK 16.11
  - Vyukov’s queue in Lthread of DPDK 16.11

Copyright 2019 FUJITSU LIMITED

Preliminary Result

Result / rte_ring

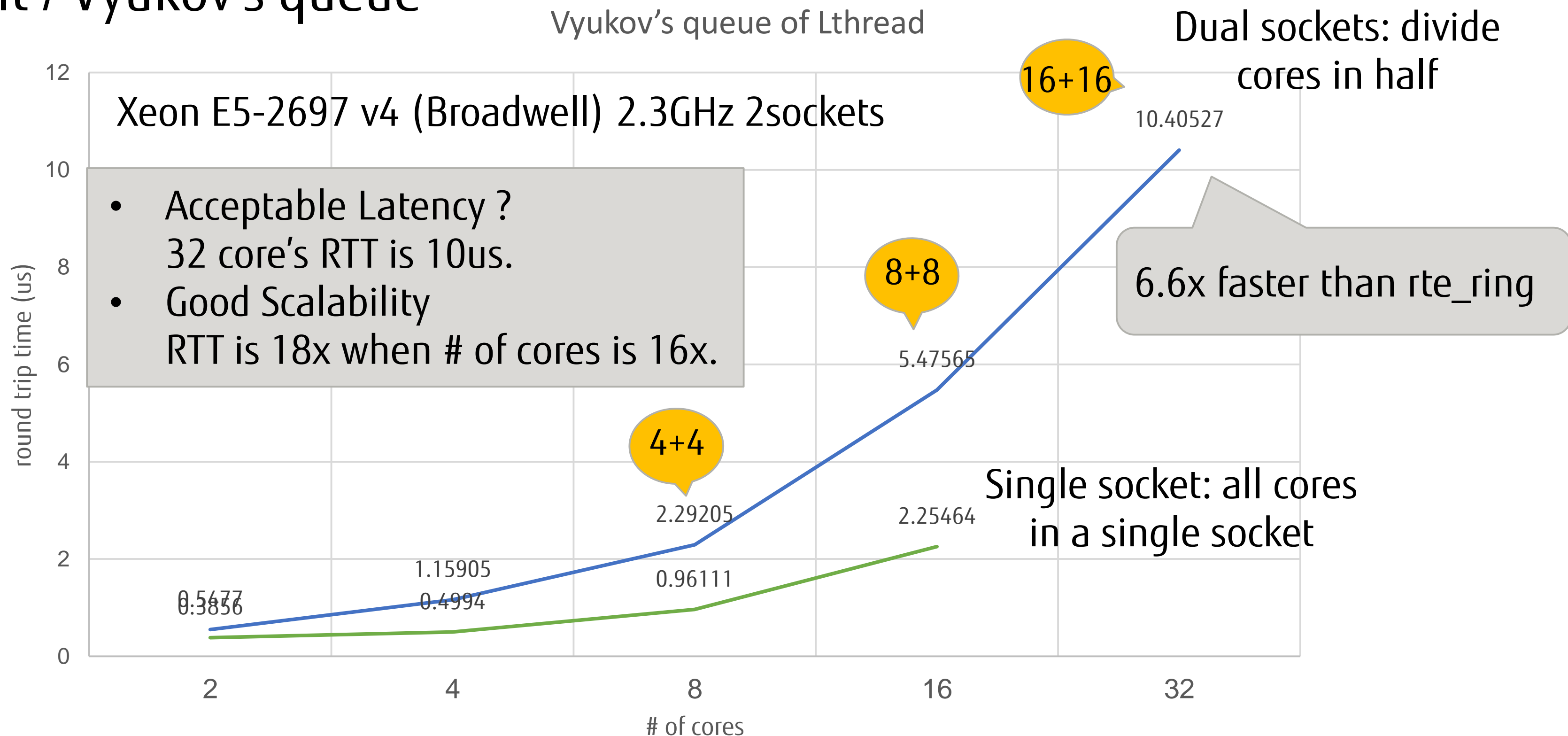


Copyright 2019 FUJITSU LIMITED

Preliminary Result (Contd.)

FUJITSU

Result / Vyukov's queue



Copyright 2019 FUJITSU LIMITED

RMI Multicore Optimizations

FUJITSU

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

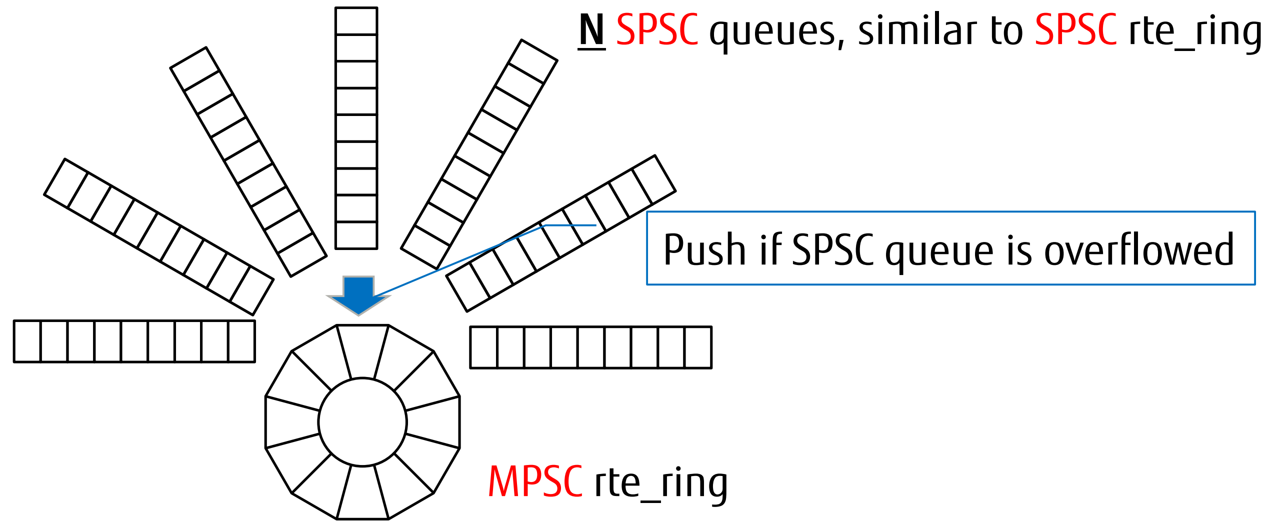
- Optimizations to reduce HW Locks, focusing on some dedicated use:
  - RMI task queue for Load Balancing
  - Lockless i-structure for HW completions
  - RMI Internode Communication (←**presented later**)

Copyright 2019 FUJITSU LIMITED

RMI task queue

FUJITSU

- 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

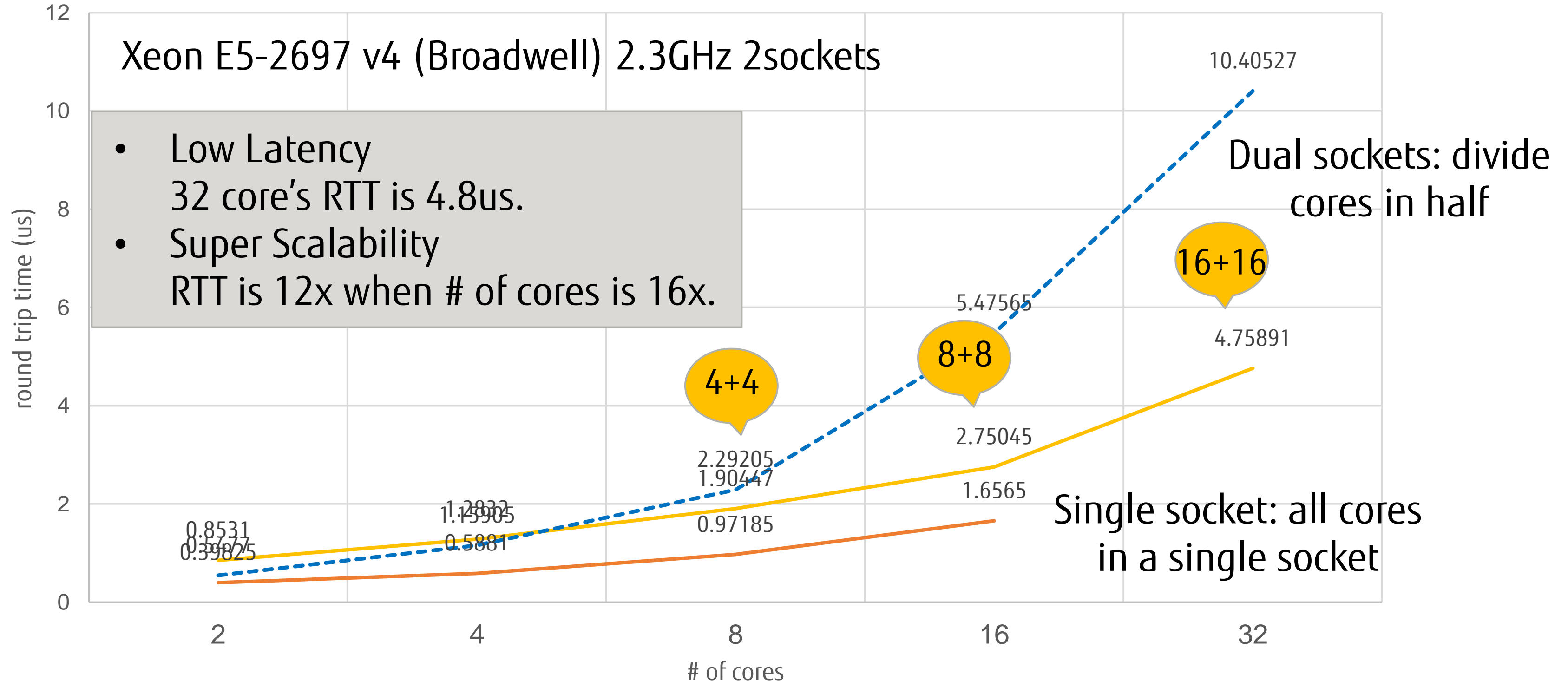


Copyright 2019 FUJITSU LIMITED

RTT Benchmark Result

FUJITSU

Result / RMI task queue



Copyright 2019 FUJITSU LIMITED

Prime Benchmark

FUJITSU

- 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

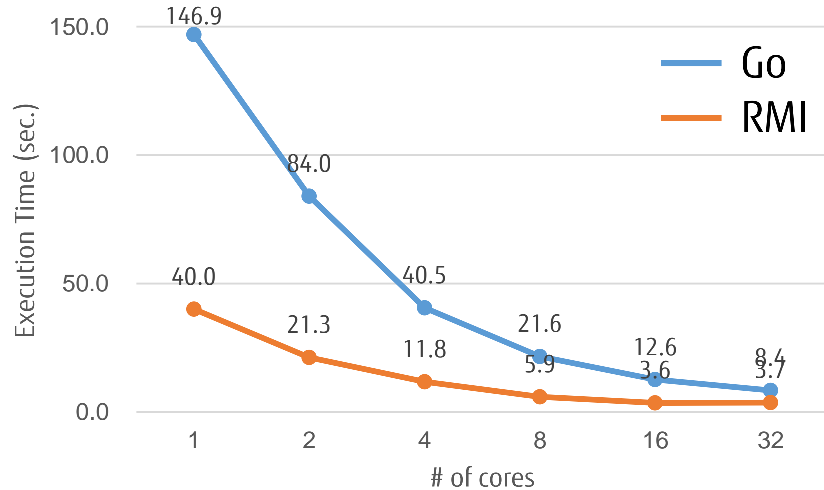
Copyright 2019 FUJITSU LIMITED

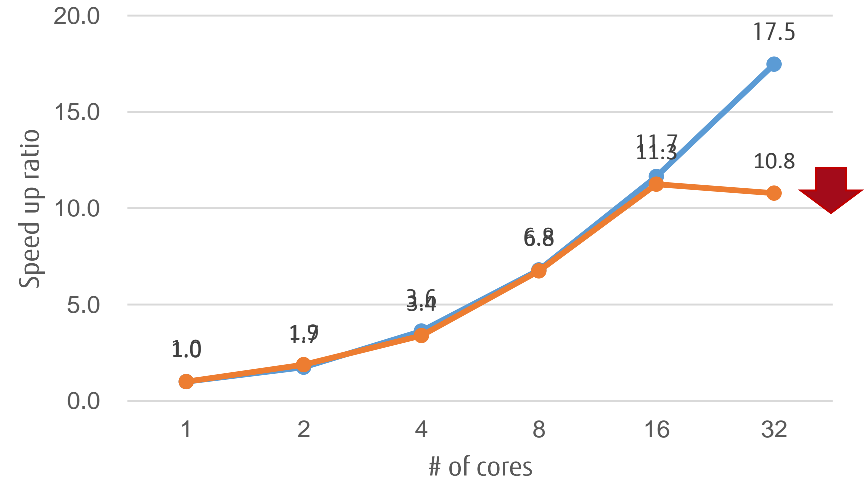
Prime Benchmark Result

FUJITSU

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

SUPERMICRO 2028U-TN24R4T / Xeon E5-2697 v4 2.3GHz 36 Cores / 512GB





Copyright 2019 FUJITSU LIMITED

RMI Internode Communication

Design of Communication Primitives

FUJITSU

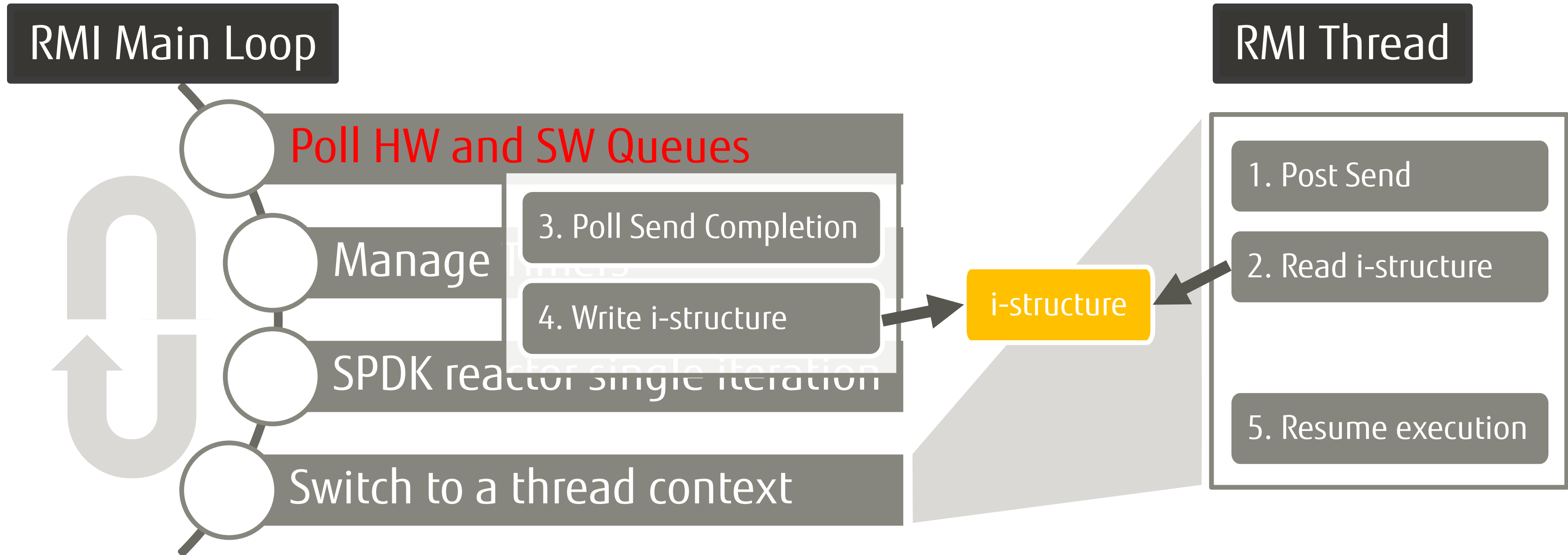
- Verbs API is commonly used in RDMA over Converged Ethernet (RoCE).
  - We adopt RoCE for low latency and low CPU usage.
- Communication primitives for RMI
  - Using native Verbs API?
    - Verbs API provides asynchronous posting and polling style primitives
      - E.g. `ibv_post_send()` and `ibv_poll_cq()`
    - 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()`

Copyright 2019 FUJITSU LIMITED

Implementation of Blocking Primitives

FUJITSU

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

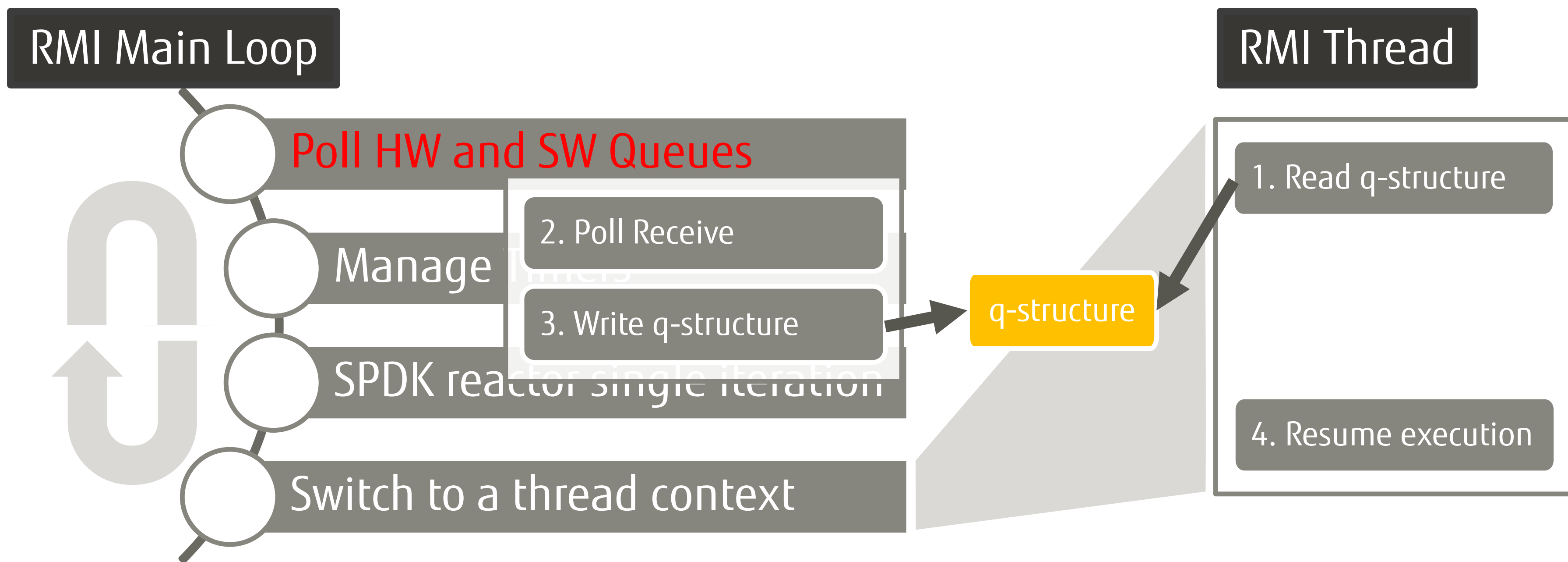


Copyright 2019 FUJITSU LIMITED

Implementation of Blocking Primitives (Contd.)

FUJITSU

- Receive
  - q-structure is used for synchronization as multiple messages may arrive.

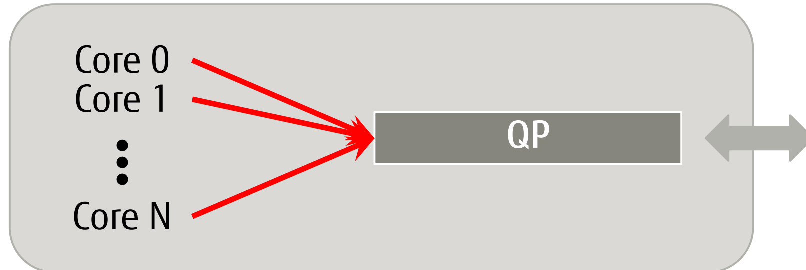


Copyright 2019 FUJITSU LIMITED

Multicore Performance Issue of Internode Comm.

FUJITSU

- 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

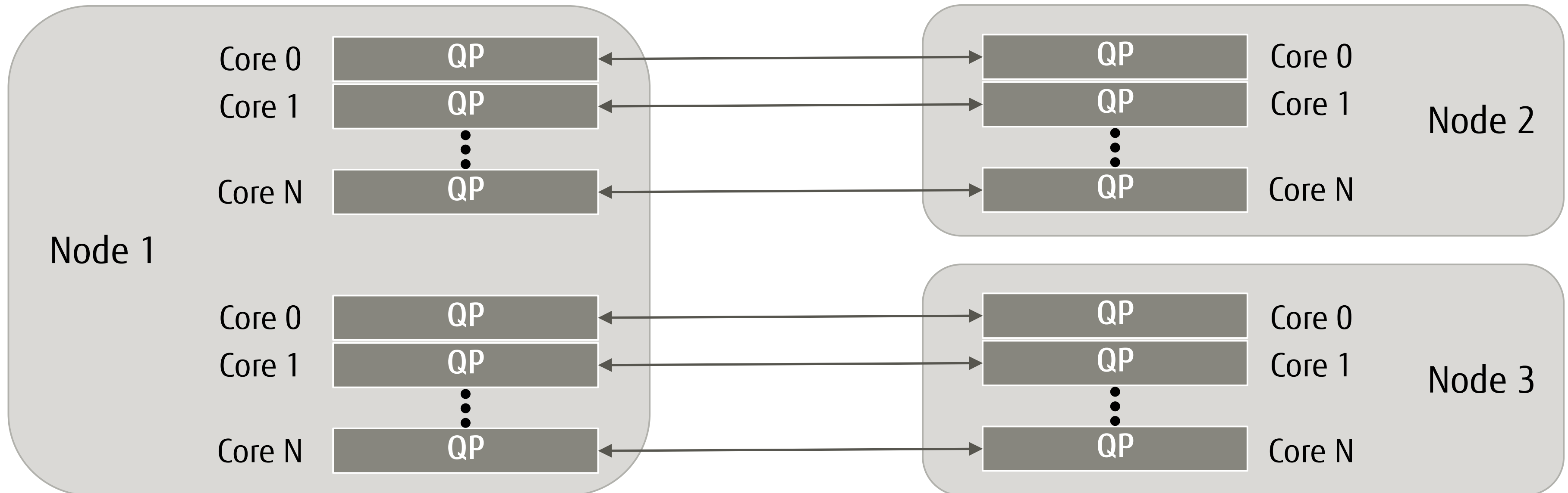


Copyright 2019 FUJITSU LIMITED

Dedicated QP for Each Core

FUJITSU

- 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.



Copyright 2019 FUJITSU LIMITED

Dedicated QP for Each Core (Contd.)

FUJITSU

- 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 $= \text{Max. \# of destination nodes} \times \text{\# of RoCE ports} \times \text{\# of cores}$
      $= 15 \times 2 \times 40 = 1200$

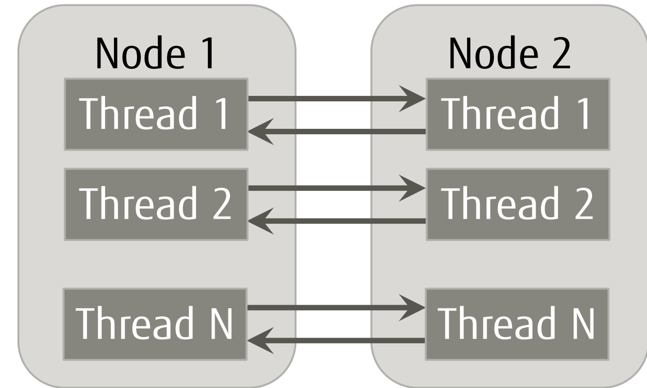
We have enough QPs for our case.

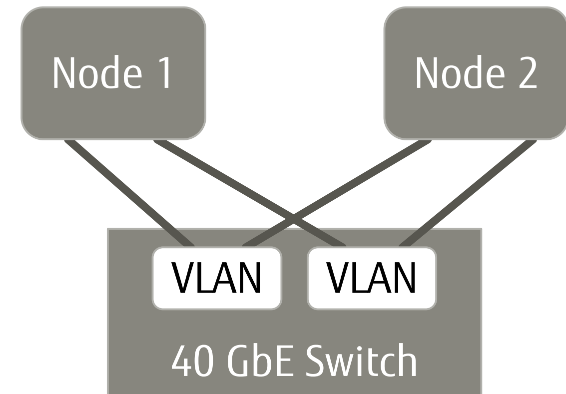
Copyright 2019 FUJITSU LIMITED

Performance Evaluation Environment

FUJITSU

- **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



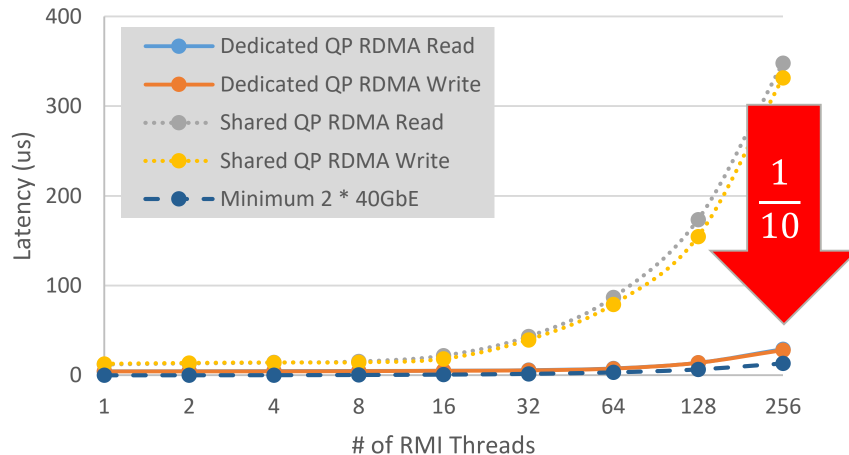


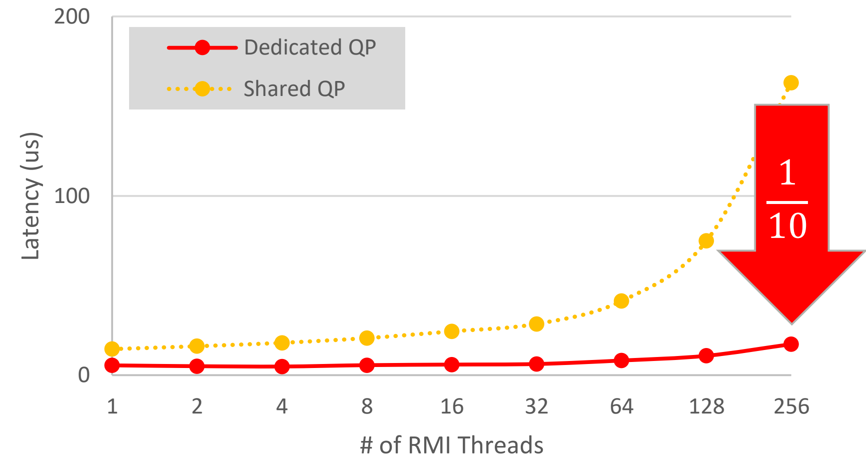
Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results

FUJITSU

- 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.



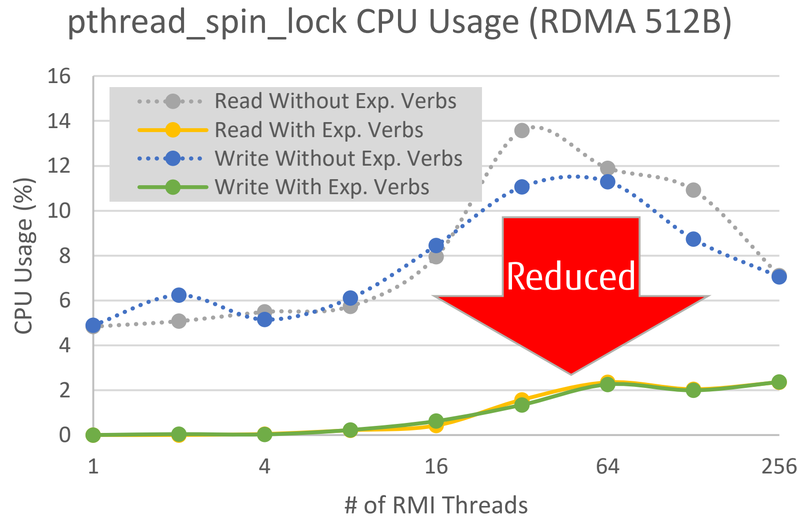


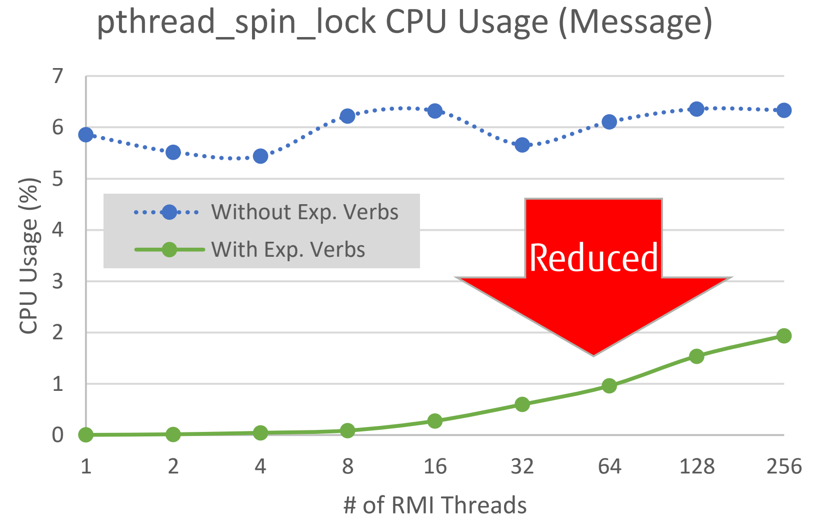
Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results (Contd.)

FUJITSU

- 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()`.





Copyright 2019 FUJITSU LIMITED

Summary

FUJITSU

- 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

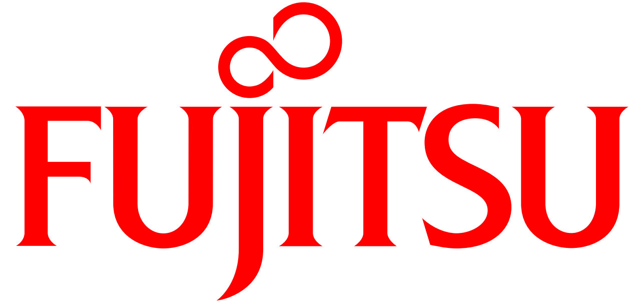
Copyright 2019 FUJITSU LIMITED

Questions ?

FUJITSU

- E-mail: flab-spdk-dev@ml.labs.fujitsu.com

Copyright 2019 FUJITSU LIMITED



shaping tomorrow with you

Bdev Extension

FUJITSU

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

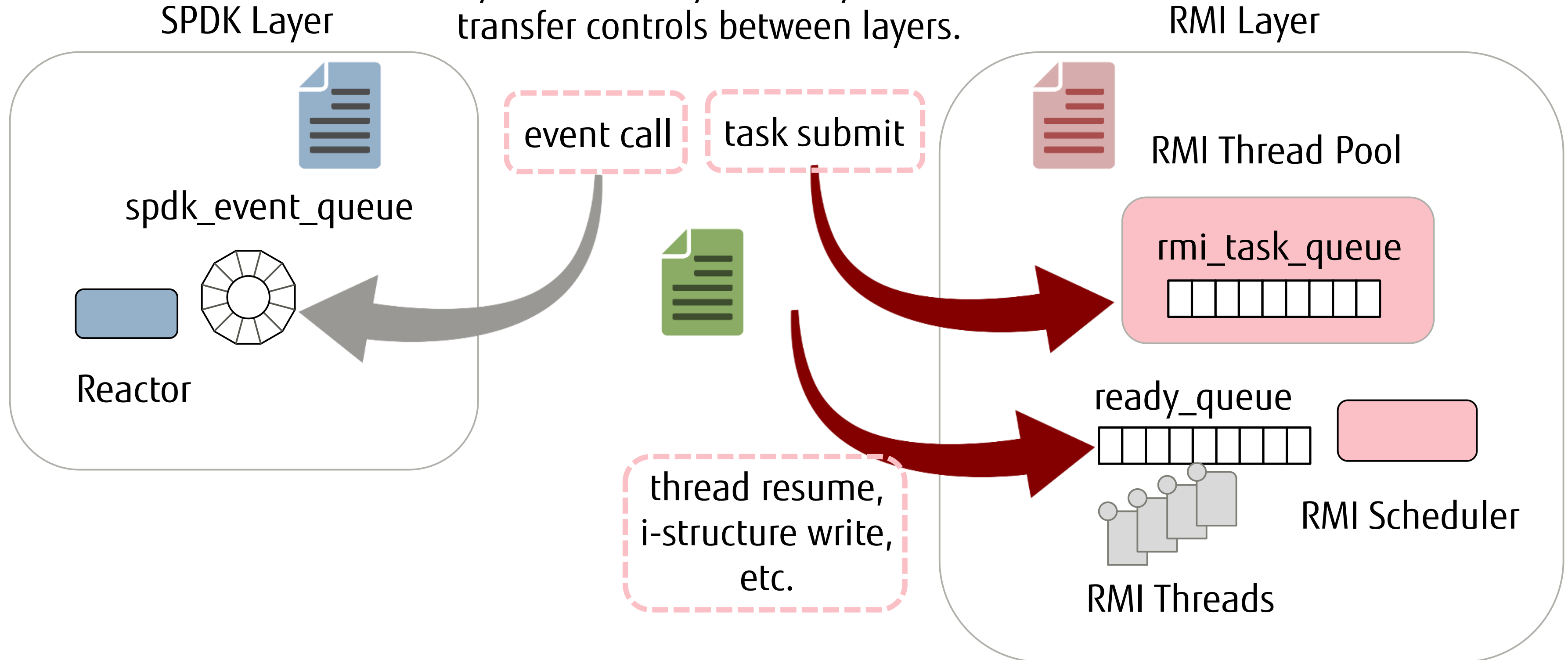
RMI design

- 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

Copyright 2019 FUJITSU LIMITED

Control Transfer between RMI and SPDK

FUJITSU

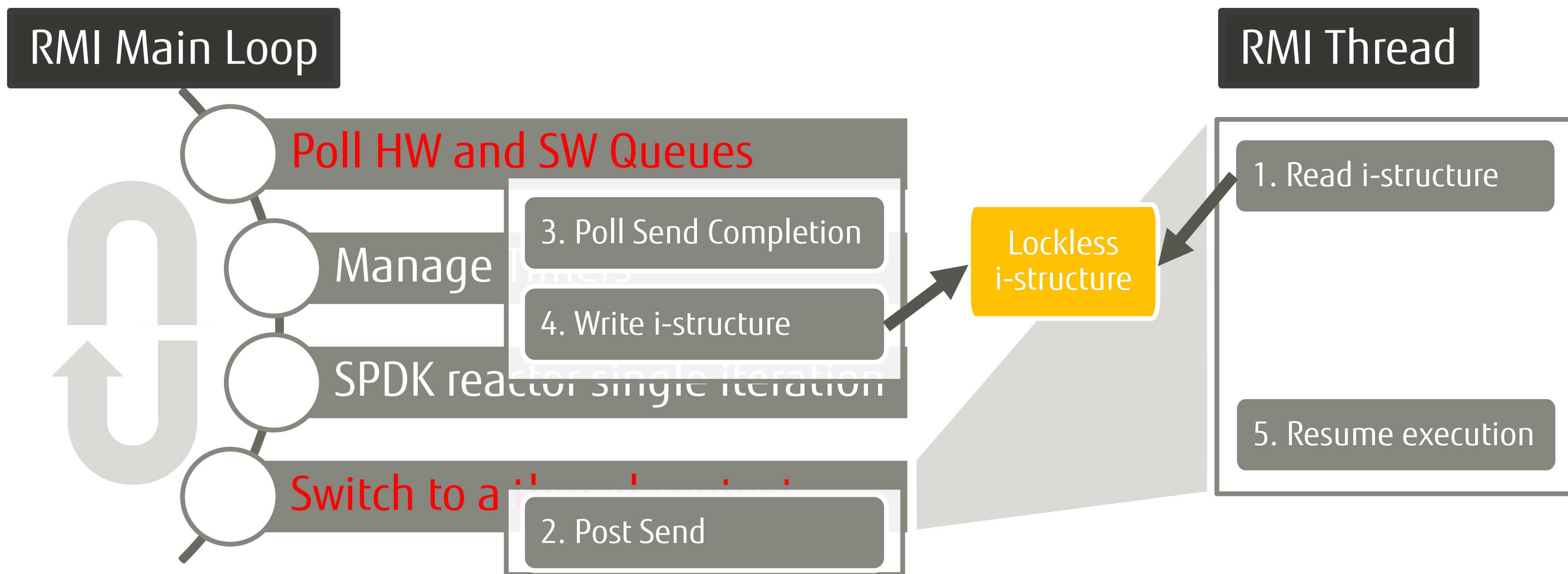


Copyright 2019 FUJITSU LIMITED

Lockless i-structure

FUJITSU

- The HW lock in i-structure can be safely removed if posting Send is performed AFTER the RMI Thread is blocked.



Copyright 2019 FUJITSU LIMITED

Lockless i-structure (Contd.)

FUJITSU

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.

```
...
                  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.

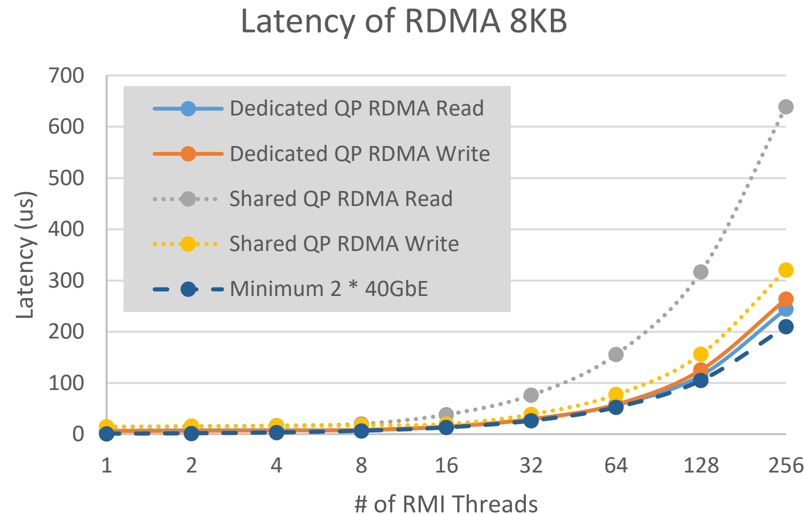
```
static void pending_send(rmi_istr_int *istr, void *arg) {
    // Executed in the scheduler context just after the RMI
                        // Thread is blocked.
 post_send(...);
}

...
  int ret = rmi_istr_lockless_int_read(&istr,
                                             pending_send, &args);
               // Blocks until the Send is completed.
```


Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results (Contd.)

FUJITSU

- Latency of RDMA 8KB is limited by the 40GbE throughput.

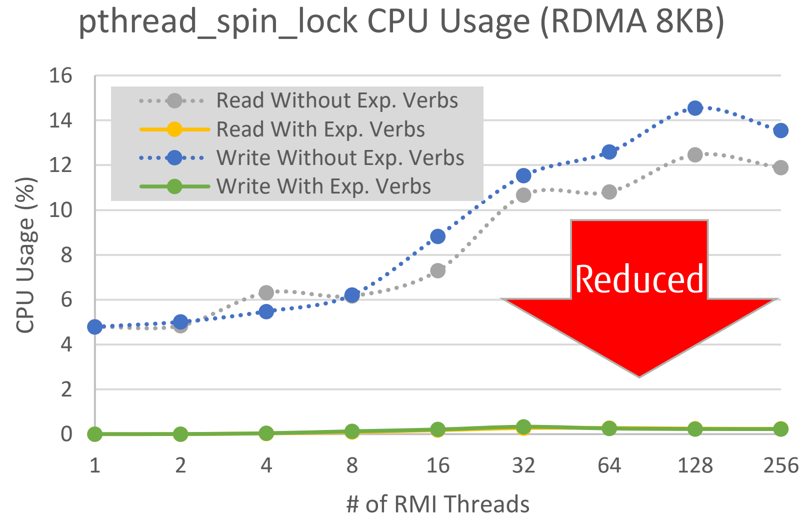


Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results (Contd.)

FUJITSU

- 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()`.

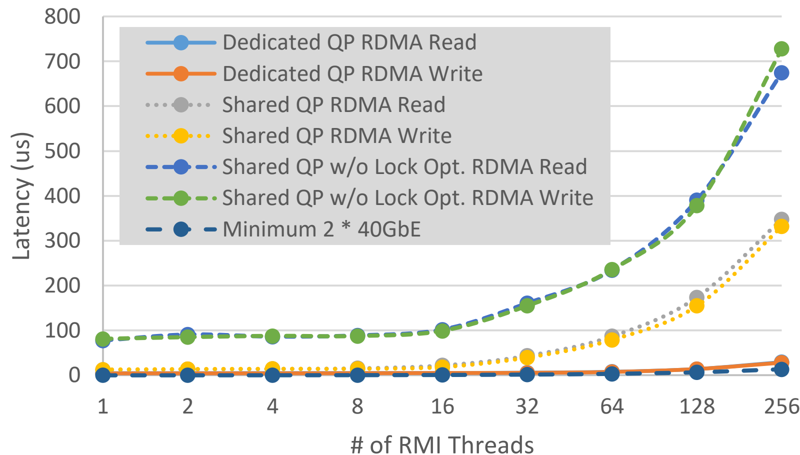


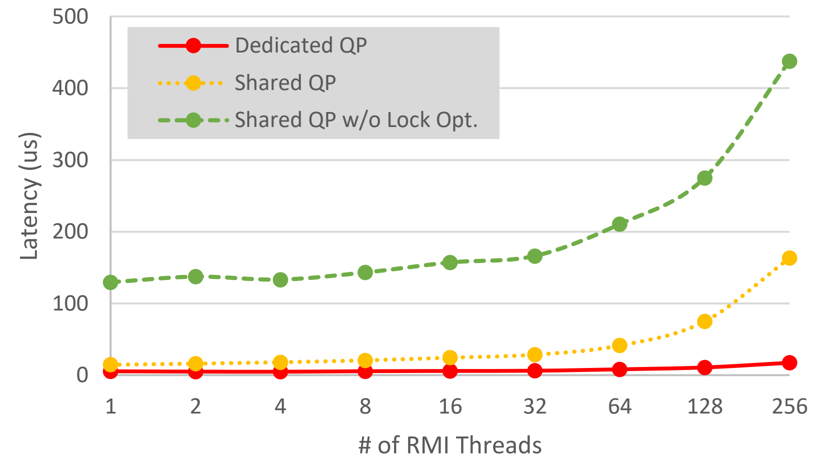
Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results (Contd.)

FUJITSU

- Shared QP without Lock Optimization has severe Performance Degradation.



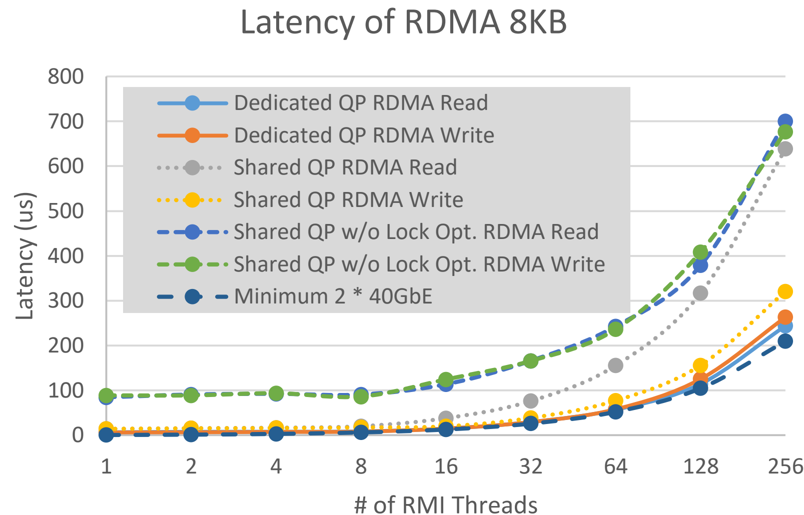


Copyright 2019 FUJITSU LIMITED

Performance Evaluation Results (Contd.)

FUJITSU

- Shared QP without Lock Optimization has severe Performance Degradation.



Copyright 2019 FUJITSU LIMITED