A Case for Early and Informed Request Scheduling

Kostis Kaffes
Platform Lab Seminar
October 8th, 2019
Intra-Server Scheduling
with Jack Humphries, David Mazières, and Christos Kozyrakis
Request Scheduling

Our goal is to develop a system that allows the scheduling of latency-critical requests on the cores of a single server while satisfying service-level objectives.

Requirements
- Low tail latency at the microsecond scale
- High throughput
- High efficiency
- Support for highly-variable execution times

KV get/put
SLO: 99% 50us @100kRPS

KV Range Query
SLO: 99% 500us @5kRPS
Meeting these requirements is hard

**Problem:** High OS and software overheads

**Solutions:** User-space network stacks and eliminating scheduling

**Problem:** Queue imbalances

**Problem:** Head-of-line blocking

Fixing these problems requires:

- Centralized queue
- Preemption
- Scheduling policies tailored for each workload
Solution: Shinjuku [NSDI’19]

A single address-space operating system that achieves microsecond-scale tail latency for all types of workloads regardless of variability in task duration

Key Features:
• Dedicated core for scheduling and queue management
• Leverage hardware support for virtualization for fast preemption
• Match scheduling policy to task distribution and target latency
Shinjuku Design

Results

- On-par with state-of-the-art for light-tailed workloads
- Up to 88% lower tail latency and 6x higher throughput for heavy-tailed workload

Steps:
1. Process packets and generate application-level requests
2. Pass requests to centralized dispatcher using shared memory
3. Add requests to centralized queue
4. Schedule requests to worker cores using shared memory
5. Send replies back to clients through the networking subsystem
6. Interrupt long running requests and schedule other requests from the queue

Problems:
- Dispatcher can be the bottleneck
- Excessive data movement between cores
- Wastes cores for polling even at low load

Results:
- On-par with state-of-the-art for light-tailed workloads
- Up to 88% lower tail latency and 6x higher throughput for heavy-tailed workload
Opportunity: NICs are not “dumb” anymore

By offloading the dispatcher and networking subsystem to a SmartNIC we can:

→ Potentially process requests at line rate
→ Steer the requests directly to the core where they will execute
→ Save host cores for other applications
SmartNIC Scheduling Requirements

1. Connection termination and application request parsing
2. Request steering to host cores
3. Store system state and request queues
4. Ability to interrupt execution on host cores
NIC Types

1. Fixed function (Segmentation, Checksum Calculation, RSS)
   • No flexibility

2. Programmable Pipeline, e.g., P4 (Netronome Agilio CX)
   • Somewhat flexible
   • Processes packets at line rate
   • Cannot store a request queue

3. ARM cores that run a regular OS (Mellanox Bluefield, Broadcom Stingray)
   • Full flexibility and state/queue handling
   • Hard to achieve line rate
Shinjuku Offload [HotNets’19] on SmartNIC

Offloaded the dispatcher and networking subsystem to a Broadcom Stingray ARM-based SmartNIC.
Shinjuku Offload Design

1. Steer packets to ARM cores and generate application-level requests
2. Pass requests to different dispatcher ARM core
3. Send requests to host worker cores through NIC VFs
4. Use local timer to preempt after time slice expires
5. Notify dispatcher that request execution finish and send reply to client
Challenge: Host-NIC communication latency

Problem: High communication latency (2.56us) between ARM and host cores leads to low worker utilization

Solution:
→ Enqueue multiple requests on each worker core to hide the latency
→ Preemptive scheduling ensures bounded latency and avoid head-of-line blocking
Challenge: Preempting worker cores

Problem:
→ The NIC cannot send low-overhead “posted” interrupts to host cores
→ 100% overhead compared to vanilla Shinjuku
→ NIC-host communication latency worsens the situation

Solution:
→ Use local timer interrupts
→ ”Post” them directly to user-space
→ Overall cost: 1312 cycles
Evaluation

Systems

Shinjuku
Shinjuku Offload

Synthetic benchmark with different service time distributions

Hardware

2.3Ghz Intel E5-2658 processor
Shinjuku NIC: Intel 82599ES 10Gb
Shinjuku Offload NIC: Broadcom Stingray PS225 – 8 ARMv8 A72 cores
Shinjuku-Offload at low core count

NIC offload allows us to achieve much higher throughput for the same number of host cores used.
Can we hide the communication latency?

We need at least 3-5 outstanding requests to hide the communication latency.
Shinjuku-Offload at high core count

Offloaded dispatcher cannot scale to more than 1.5 MRPS due to:
- Slower ARM CPU
- High communication overhead

Better

16 host cores

Fixed 1us

Throughput (100 kRPS)

99% latency
How to overcome SmartNIC limitations

1. No hardware support for line-rate scheduling
   **Proposal:** Add ability to queue packets/requests in a programmable pipeline

2. High communication overhead between NIC and host cores
   **Proposal:** Replace PCIe with some low latency coherent interconnect, e.g., CXL

3. Lack of low-overhead NIC-to-host interrupts
   **Proposal:** Allow the NIC to issue user-level interrupts
Inter-Server Scheduling
with Hang Zhu, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin
Problem: Scheduling to multiple servers

Assumption: All requests can be served by all servers
Common Approach: Random Choice

Randomly choose a server to send the request to

- No scheduling overhead
- Queue Imbalances
- Head-of-line blocking
Ideal: Centralized Scheduler

Scheduler-side queuing – Schedule only when there is available capacity

- Perfect load balancing
- No head-of-line blocking
- Hard to scale
- Requires queueing at scheduler side
Practical: **Rack-Scale Centralized Scheduling**

Implement centralized scheduling logic at P4 pipeline at ToR switch. *Join Shortest Queue* to avoid scheduler-side queueing.

- Switch ensures line-rate scalability
- Near-optimal load balancing, robust to service-time variability
Challenge: Tracking Server Queue Length

Option 1: Switch periodically probes server queue lengths
-- Too coarse-grain for microsecond scale

Option 2: Use switch counters to proactively keep track of server load
-- Susceptible to packet loss at high load

Option 3: Piggyback queue length to replies ✅
Challenge: Herding

Join Shortest Queue is imperfect:
- Delayed update of server queue lengths at the switch
- Multiple consecutive requests sent to the same server

Solution: Randomness
- Join Shortest of K Queues
RSCS requires minimal changes

- Compatible with existing L2/L3, transport, and RPC protocols
- Adds only a 3-field header
ToR Implementation

- **ReqTable**
  
  Request id → Server

- **LoadTable**
  
  Server Queue Lengths
Evaluation

Systems

*RSCS* with per-node *Shinjuku*

*Random* with per-node *Shinjuku*

Synthetic benchmark with different service time distributions

RocksDB - in-memory database

Hardware

6.5Tbps Barefoot Tofino switch w/ 8 8-core servers
Is RSCS better than random choice?

RSCS achieves 20% higher load for a 99% latency target of 1000us.
How does RSCS scale?

RSCS achieves linear scaling

90% - 50us
10% - 500us

Better
How does a real workload behave?

RocksDB
90% - GET
10% - SCAN

RSCS achieves 25% higher load for a 99% latency target of 1000us
Does the queue length tracking method matter?

Piggybacking outperforms switch tracking because of losses/retransmissions.
Is there a herding effect?

Big gap at lower loads due to herding effect
Open Questions

• Support for writes and state modification at individual servers
• Scheduling multiple (micro)services within a rack
• Going beyond a single rack
• ToR is a mini supercomputer. What else can we use it for?
Conclusion

- **Intra-server** request *scheduling* at the **NIC**
- **Inter-server** scheduling at the **ToR switch**

Early scheduling ➔ Higher efficiency and lower scheduling latency
Informed scheduling ➔ Fewer imbalances and reduced queuing