DRACONIS: NETWORK-ACCELERATED SCHEDULING FOR MICROSECOND-SCALE WORKLOADS

Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh and Samer Al-Kiswany
Microsecond-Scale Workloads

- Recent datacenter advances enable microsecond-scale workloads
  - Storage class memory, accelerators, and RDMA

- Applications are getting shorter despite accessing large datasets [1]
  - Financial analytics and algorithmic smart trading [2]
  - Realtime IoT analytics [3]
  - Rapid object detection [4]
  - Low latency web services [5]

---

Scheduling Microsecond-scale Workloads

- Low *Scheduling Tail Latency*
  - Accurate scheduling decisions

- High *Scheduling Throughput*
  - Millions of scheduling decisions per second [6]
Modern Network-Accelerated Scheduling

- Distributed queue design

- **Disadvantages:**
  - Suboptimal - *Node-level blocking*
  - Inefficient implementations
    - R2P2 \(^7\) – Recirculation
    - RackSched \(^8\) – Sampling

---

\(^7\) Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. 2019 USENIX Annual Technical Conference (ATC 19), 2019

Programmable Switches

- Challenges
  - No loops / recursion
  - Limited pipeline stages and memory
  - Access a memory register only once per packet!
  - “Switches cannot house dynamic data structures such as queues”
Draconis Overview

- A novel in-network scheduling paradigm
  - Centralized in-switch queue -> Eliminates node-level blocking
  - Supports complex scheduling policies

- Evaluation Highlights:
  - 61% lower tail latency over network-accelerated designs
  - 52x throughput over server-based designs
Outline

• Introduction
• Background
• FIFO Scheduling
• Complex Policies
• Evaluation
• Conclusion
Draconis – FIFO Scheduling

Client

Submit Job
Task 1
Task 2

Task Queue

Executor 1
Executor 2
Executor 3
Executor N
Draconis – FIFO Scheduling

Client

Task Queue

Task 1
Task 2
...

Task Retrieval

Executor 1  Executor 2  Executor 3  ...  Executor N
Draconis – FIFO Scheduling

Client

Task Queue

Executor 1
Executor 2
Executor 3
Executor N

Task 1

Task 2

...
Design Components – Task Queue

- Tasks are stored in a circular queue
- Simple yet tricky to implement on modern programmable switches

Draconis’s Task Queue Design
on_retrieve {
    rtrv_ptr = read(rtrv_ptr)
    if( queue contains a task )
    {
        // Schedule the task
        rtrv_ptr ++
    } else {
        // Queue is empty
    }
}  

**Challenge:** This accesses the pointer twice!
Design Components – Task Queue

on_retrieve {
    rtrv_ptr = read(rtrv_ptr)
    if( queue contains a task )
    {
        // Schedule the task
        rtrv_ptr ++
    } else {
        // Queue is empty
    }
}

Challenge: This accesses the pointer twice!

on_add {
    add_ptr = read(add_ptr)
    if( queue has space )
    {
        // Enqueue the task
        add_ptr++
    } else {
        // Queue is full
    }
}
Design Components – Task Queue

on_retrieve{
    rtrv_ptr = read_and_increment(rtrv_ptr);
    if(queue contains a task){
        // Schedule the task
    }
    else
        // rtrv_ptr needs fixing
}

on_add(task) {
    add_ptr = read_and_increment(add_ptr);
    if (queue is full){
        // add_ptr needs fixing
        fix(add_ptr);
    }
    if (rtrv_ptr needs fixing)
        fix(rtrv_ptr);
    // Enqueue the task
}

read_and_increment() - Optimistic atomic read and increment of pointers

fix() – Fixing pointer values as required
Design Components - Pointer Fixing

- Use packet recirculation
- Adjust pointer values as needed
  - Use Boolean flags to prevent race conditions

```c
fix(pointer){
    send_fix_pkt ( pointer, new_val )
}
```
Complex Scheduling Policies

• Classes of service
  • Priority Scheduling

• Constraint based
  • Resource-Aware Scheduling
  • Locality Scheduling
Task Swapping: Resource-Aware Scheduling

Task Retrieval
Executor_Rsrc: R2

Executor

Resources Required | Task
--- | ---
R1 | Task 1
R2 | Task 2
R3 | Task 3
... | ...

Task requires resources unavailable on executor

retrieve_ptr

Executor_Rsrc: R2

retriever_ptr

0x01
Task Swapping: Resource-Aware Scheduling

Task Swap

<table>
<thead>
<tr>
<th>Executor_Rsrc:</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Task 1</td>
</tr>
<tr>
<td>Off: 0</td>
<td>Rtrv: 02</td>
</tr>
</tbody>
</table>

Resources Required | Task
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>Task 2</td>
</tr>
<tr>
<td>R3</td>
<td>Task 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Executor

retrieve_ptr

0x02
Task Swapping: Resource-Aware Scheduling

Task Swap

<table>
<thead>
<tr>
<th>Exec_Rsrc:</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources Required</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>Task 2</td>
</tr>
<tr>
<td>R3</td>
<td>Task 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Executor

retrieve_ptr

0x02
Task Swapping: Resource-Aware Scheduling

Task Retrieval

<table>
<thead>
<tr>
<th>Exec_Rsrc:</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>Task 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resources Required</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Task 1</td>
</tr>
<tr>
<td>R3</td>
<td>Task 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Executor

retrieve_ptr

0x02
Evaluation Overview

Testbed
- 13 node cluster
  - Intel Xeon 10 core CPU with HT
  - 48GB RAM
  - 100GBps Mellanox NIC
- EdgeCore Wedge 100BF-32x
  - Intel Tofino ASIC

Workloads
- Synthetic Suite
  - Uniform Service Times (100, 250 & 500 µs)
  - Bimodal and Trimodal
  - Exponential – Mean 250 µs
- Google Cluster Traces

Evaluation Overview

Alternatives

- R2P2 [7]
- RackSched [8]
- Sparrow [9]
- Draconis-Socket-Server
- Draconis-DPDK-Server

Metrics

- 99th Percentile Scheduling Latency
- Scheduling Throughput
- Effectiveness of complex scheduling policies

Scheduling Latency

Experimental Setup:

- Synthetic workload consisting of 500 µs tasks
- Tail Latency – 99th %ile

Draconis outperforms all other alternatives by at least 61%
Concluding Remarks

- Draconis overcomes the shortcomings of modern scheduling paradigms
  - Novel in-network centralized scheduling approach
- Supports complex policies with generic design principles
  - Task swapping and Queue replication

**Evaluation Highlights:**

- **61%** lower latencies over network-accelerated scheduling
- **52x** higher scheduling throughput over server-based scheduling

- Code: [https://github.com/UWASL/Draconis](https://github.com/UWASL/Draconis)