SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel

Siyao Zhao, Haoyu Gu, Ali José Mashtizadeh

RCS Group @ University of Waterloo

USENIX ATC ’21 – July 16, 2021
The Latency Problem

- Modern server applications require low tail latency

- Recent research proposed kernel-bypass/custom dataplanes

- Problem: most applications still on traditional OSes

- Solving the latency problem in kernel is challenging
Sources of Latency

- **Cache Misses**
  - Connection migration from RSS & kernel
  - Applications do not detect any of this!
  - RPC server → 77% avoidable L2 misses
Sources of Latency

• Cache Misses
  - Connection migration from RSS & kernel
  - Applications do not detect any of this!
  - RPC server → 77% avoidable L2 misses

• Workload Imbalance
  - Over-saturated vs. under-saturated threads
  - Total processing time difference:
    ▶ Memcached 1.4% vs. GIS application 46%
• Uses kernel event facility:
  – Older: poll, select, /dev/poll
  – Newer: FreeBSD Kqueue, Linux epoll, Windows IOCP
  – 20+ years old

• Maximum visibility into OS and applications

• Application workflow:
  – Applications register events to kernel event objects
  – Worker threads poll/listen on kernel event object
Event-Driven Programming Models

<table>
<thead>
<tr>
<th>Model</th>
<th>1:1</th>
<th>1:N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Object</td>
<td>Private</td>
<td>Shared</td>
</tr>
<tr>
<td>Scalability</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Schedulability</td>
<td>Poor</td>
<td>Good</td>
</tr>
<tr>
<td>Affinity</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Popularity</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

The 1:1 model

The 1:N model
Our System: Scheduatable Kqueue (SKQ)

- Efficient event scheduling on top of Kqueue
- Multicore scalability in the 1:N model
- Event scheduling to reduce latency
  - Cache misses & workload imbalance
- Event delivery control
  - Event prioritization & event pinning
Main thread

...  
// Create a single SKQ instance
int skq = kqueue();

// Set scheduling policy
int sched = KQ_SCHED_CPU;
ioctl(skq, FKQMULTI, &sched);
...

Worker threads

while (true) {
    const int maxev = 32;
    struct kevent evs[maxev];

    // Query events
    nev = kevent(skq, NULL, 0, &evs, maxev, NULL);

    // Process events
    ...
}
SKQ Architecture

- **Scalability**
  - Thread-private event queue *kevq*
  - Fine-grained locking

- **Event scheduling and delivery control**
  - The event scheduler

- **Best of both worlds via scheduling**
  - 1:N model to applications
  - 1:1 model internally
Scalability Showdown: SKQ vs. Kqueue

- POSIX pipes instead of sockets
- 1:N SKQ scales linearly
- 1:N Kqueue sees bottleneck
Performance Improvement: SKQ vs. Kqueue

- Memcached with 1:1 model
- Facebook Mutilate workload
- −33% tail latency at peak

**Benchmark Baseline**

We use 1:1 SKQs throughout the talk to isolate the benefit of scheduling.
Scheduling Policies

- Cache locality policies
- Load balancing policies
- Hybrid policies
Challenges of Efficient Event Scheduling

- Little overhead available for scheduling
  - Millions of events per second

- CPU cost, lock contention, and cache footprint
  - Statistics
  - Data structures

- Memcached:
  - 15k cycles per request (amortized)
  - L3 cache miss ~ 400 cycles
  - 2.7% drop in throughput
Cache Locality Policies

- **CPU affinity**
  - Delivers to the **triggering** core
  - Follows connection migration
  - Cache locality in the networking stack

- **Queue affinity**
  - Pins to the **first** core
  - Cache locality in userspace
Cache Locality Policies in Memcached

- A uniform workload
- Cache locality dominates latency
- Facebook Mutilate workload
  - +9% low-latency throughput
  - −26% tail latency at peak
CPU Affinity vs. Queue Affinity

- L2 cache misses in a RPC server
- Uniform workload, Memcached-like
- CPU affinity –31.2% L2 cache misses

<table>
<thead>
<tr>
<th>Policy</th>
<th>TCP input</th>
<th>TCP output</th>
<th>Event activation</th>
<th>Event query</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>252k</td>
<td>15k</td>
<td>63k</td>
<td>166k</td>
<td>496k</td>
</tr>
<tr>
<td>Queue</td>
<td>343k</td>
<td>33k</td>
<td>95k</td>
<td>250k</td>
<td>721k</td>
</tr>
<tr>
<td>Vanilla</td>
<td>828k</td>
<td>76k</td>
<td>45k</td>
<td>1235k</td>
<td>2184k</td>
</tr>
</tbody>
</table>
Imbalanced Workload: RocksDB

- Facebook ZippyDB workload
- Imbalanced – slow SEEK requests
- Cache locality policies don’t help
Load Balancing Policies

- **Best of two**
  - Selects the better of two random kevqs
  - Statistics keeping
    - Number of events
    - Average processing time per event
Load Balancing Policies

- **Best of two**
  - Selects the better of two random key values
  - Statistics keeping
    - Number of events
    - Average processing time per event

- **Work stealing**
  - Idle threads steal work
  - Minimal interference

- **Best of two + work stealing**
Hybrid Policies

- Load balancing + cache locality
  - Best of two vs. cache-local kevq
  - Cache miss penalty

- Hybrid Policy
  - CPU affinity + best of two + work stealing
  - 27.4× low-latency throughput
  - Up to 1022× lower tail latency
SKQ vs. Kernel Bypass: Uniform 10 µs Workload

- RPC server
  - Uniform
  - 10 µs request service time
  - CPU affinity policy

- Compared 150 µs tail latency

- Shenango
  - 1.67× low-latency throughput

- Bottleneck: system call overhead
SKQ vs. Kernel Bypass: Uniform 20 µs Workload

- RPC server
  - Uniform
  - 20 µs request service time
  - CPU affinity policy

- Compared 150 µs tail latency

- Shenango
  - 1.5× low-latency throughput

- Bottleneck: system call overhead
SKQ vs. Kernel Bypass: Zipf-like Workload

- RPC server
  - Zipf-like
  - 20.5 µs average service time
  - Hybrid policy

- Compared 150 µs tail latency

- Shenango
  - 1.17x low-latency throughput

- Benefit from event scheduling
• Kernel bypass:
  - Short and uniform

• Kernel event scheduling:
  - Long and imbalanced

• Unfair comparison
  - Different OSes
  - Different TCP stack
Policy Selection Guidelines

- Uniform workloads
  - CPU affinity

- Imbalanced or IO-heavy workloads
  - Hybrid policy (CPU affinity + Best of two + Work stealing)
Conclusion

• A practical solution to the latency problem

• More optimization opportunities in kernel

• See paper for:
  - Event prioritization and pinning
  - Cache miss analysis and more benchmarks
  - Design details and optimizations

• Source code available:
  - https://rcs.uwaterloo.ca/skq/