Be Fast, Cheap and in Control with SwitchKV

Xiaozhou Li
Goal: fast and cost-efficient key-value store

- Store, retrieve, manage key-value objects
  - Get(key)/Put(key,value)/Delete(key)

- Target: cluster-level storage for modern cloud services
  - Need both high-performance and low-cost operations

- Workload characteristics
  - large datasets
  - small objects
  - random access
  - massively parallel
  - deeply skewed
  - rapidly changing
Goal: fast and cost-efficient key-value store

- Use DRAM and flash efficiently
  - meet performance and capacity goals with min cost

SwitchKV: hash partitioned flash-based storage cluster

- Ensure dynamic load balancing
  - always fully utilize all the storage servers

- Achieve maximum parallelism in multi-core systems
  - fully utilize the processing power of modern CPUs

SwitchKV: exploit fast programmable networking
Key challenge: dynamic load balancing

- Most servers are either over- or underutilized.
  - low throughput and high tail latency

- Today’s solutions: data migration / replication
  - high system overhead, consistency challenge
Fast, *small* cache can ensure load balancing

Need only cache $O(n \log n)$ items to provide good load balance, where $n$ is the number of backend nodes. [Fan, SOCC’11]

E.g., 100 backends with hundreds of billions of items + cache with 10,000 entries

• How to efficiently serve queries with cache and backend nodes?
• How to efficiently update the cache under dynamic workloads?
High overheads with traditional caching architectures

- Cache must process all queries and handle misses
- In our case, cache is small and hit ratio could be low
  - Throughput is bounded by the cache I/O
  - High latency for queries for uncached keys
SwitchKV: content-aware routing

Switches route requests directly to the appropriate nodes

- Latency can be minimized for all queries
- Throughput can scale out with # of backends
- Availability would not be affected by cache node failures
Exploit SDN and switch hardware (NSDI’16)

- Clients encode key information in packet headers
  - Encode **key hash in MAC** for read queries
  - Use destination **backend IP** for all queries
- Switches maintain forwarding rules and route query packets

![Flowchart Diagram]

- Packet In → L2 table:
  - **exact match rule per cached key**
    - If hit, Packet Out to the cache
    - If miss, TCAM table:
      - **match rule per physical machine**
        - Packet Out

---

8
Frequently asked questions at NSDI’16:

- “L2 is so fundamental, can you really change it?”
- “Is it realistic for production systems?”
- “So I cannot use regular UDP socket any more?”
- “Any side effects to other network apps?”

Good news: P4 switches

- Fully user-programmable data plane
- No dangerous L2/L3 header changes by the application
P4 switch pipeline

- **Parser**
  - Converts packet data into metadata (Parsed Representation)

- **Mach-Action Units (MAU)**
  - Operate on metadata

- **Metadata Bus**
  - Carries information within the pipeline

- **Deparser**
  - Converts metadata back into a serialized packet
New SwitchKV packet for P4 switch

- **IP**: all queries use backends as destination
- **UDP**: reserve port # for the key-value storage service
- **Type**: read, write, delete, miss
- **Key Hash**: 64b or 128b hash value of the key
P4 program for SwitchKV

```p4
header_type kv_t {
    fields {
        type: KV_TYPE_BITWIDTH;
        key_hash: KV_HASH_BITWIDTH;
    }
}
header kv_t kv

control ingress {
    ...
    if (valid(kv) and kv.type == READ) {
        apply (cache_lookup);
    }
    ...
}

table cache_lookup {
    reads {
        kv.key_hash: exact;
    }
    actions {
        cache_hit; // change dst Eth & IP
        _no_op;
    }
    size: CACHE_SIZE;
}
```

```p4
parser parse_udp {
    extract (udp);
    return select (latest.dstPort) {
        UDP_PORT_KV: parse_kv;
        default: ingress;
    }
}
parser parse_kv {
    extract (kv);
    return ingress;
}
```
Keep cache and switch rules updated

• New challenges for cache updates
  ▪ Only cache the hottest $O(n \log n)$ items
  ▪ Limited switch rule update rate

• Goal: **react quickly** to workload changes with **minimal updates**

---

**Diagram:**
- Switch rule update
- Controller
- Top-$k$ <key, load> list (periodic)
- Fetch <key, value>
- Bursty hot <key, value> (instant)
Goal: fast and cost-efficient key-value store

- Use DRAM and flash efficiently
  ➔ meet performance and capacity goals with min cost

SwitchKV: hash partitioned flash-based storage cluster

- Ensure dynamic load balancing
  ➔ always fully utilize all the storage servers

- Achieve maximum parallelism in multi-core systems
  ➔ fully utilize the processing power of modern CPUs

SwitchKV: exploit fast programmable networking
Multicore parallelism

• Concurrent access
  - more generic abstractions
  - require careful algorithm engineering

• Exclusive access
  - no inter-core communication
  - load imbalance may hurt performance
  - need to deliver the requests to the right cores
SwitchKV servers: exclusive access

- Use Intel DPDK to deliver each query to the right core
  - match on (self-defined) packet header fields
SwitchKV servers: exclusive access

- Use Intel DPDK to deliver each query to the right core
  - match on (self-defined) packet header fields
- Each core runs its own load tracker and counter
  - easy to ensure high performance and correctness
- Avoid performance drop due to load imbalance
  - backends do not face high skew in key popularity
  - cache nodes exploit CPU caches and packet burst I/O
Evaluation (NSDI’16)

- How well does a fast small cache improve the system load balance and throughput?
- Does SwitchKV improve system performance compared to traditional architectures?
- Can SwitchKV react quickly to workload changes?
Reference backend

- 1 Gb link
- Intel Atom C2750 processor
- Intel DC P3600 PCIe-based SSD
- RocksDB with 120 million 1KB objects
- 99.4K queries per second
### Evaluation Platform

**Client**
- 40 GbE
- Xeon Server 1

**Cache**
- 40 GbE
- Ryu
- Pica8 P-3922 (OVS 2.3)

**Backends**
- 40 GbE
- Xeon Server 3

**Backends**
- 40 GbE
- Xeon Server 4

<table>
<thead>
<tr>
<th># of backends</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend tput</td>
<td>100 KQPS</td>
</tr>
<tr>
<td>keyspace size</td>
<td>10 billion</td>
</tr>
<tr>
<td>key size</td>
<td>16 bytes</td>
</tr>
<tr>
<td>value size</td>
<td>128 bytes</td>
</tr>
<tr>
<td>query skewness</td>
<td>Zipf 0.99</td>
</tr>
<tr>
<td>cache size</td>
<td>10,000 entries</td>
</tr>
</tbody>
</table>

**Default settings in this talk**

- Use Intel DPDK to efficiently transfer packets and modify headers
- Client adjusts its sending rate, keep loss rate between 0.5% and 1%
Throughput with and without caching

- Cache (10,000 entries)
- Backends aggregate (without cache)
- Backends aggregate (with cache)
- Cache (10,000 entries)

Throughput (MQPS)

Workload Distribution

- uniform
- zipf-0.9
- zipf-0.95
- zipf-0.99
Throughput vs. Number of backends

backend rate limit: 50KQPS, cache rate limit: 5MQPS
End-to-end latency vs. Throughput

![Graph showing latency vs. throughput for different methods, including Look-aside, Look-through, and SwitchKV, with throughput limit and 99th percentile highlighted.]

- Throughput limit (beyond which >1% of packets are dropped)
Throughput with workload changes

Make 200 cold keys become the hottest keys every 10 seconds
SwitchKV: fast and cost-efficient KV store

- Fast, small cache guarantees backend load balancing

- Efficient **content-aware programmable network switching**
  - Low (tail) latency
  - Scalable throughput
  - High availability

- Keep high performance under highly dynamic workloads
Thanks!