RobinHood: Tail Latency-Aware Caching
Dynamically Reallocating from Cache-Rich to Cache-Poor

Daniel S. Berger (CMU)

Joint work with:
Benjamin Berg (CMU), Timothy Zhu (PennState),
Siddhartha Sen (Microsoft Research), Mor Harchol-Balter (CMU)

To appear at USENIX OSDI (October 2018).
Microsoft Web Architecture

Goal: minimize 99-th percentile request latency (P99)

Request must wait for last query!

Goal: minimize 99-th percentile request latency (P99)
What Causes High P99 Request Latency?

Observations at xbox.com (3/2018):

Better load balancing?

Elastically scale backends?

Already implemented!
What Else Can We Do?

Can we use the aggregation cache to reduce P99 request latency?

Aggregation Cache
Currently shared among queries to all backends

Observations for xbox.com (3/2018):

Query P99 Latency [Normalized]

Hour of the Day

Products
Ads
Recom.
Can We Use Caching to Reduce the P99?

Belief: No

The Tail at Scale

“Caching layers do not directly address tail latency, aside from configurations where the entire working set can reside in a cache.”

State-of-the-art caching systems focus on hit ratio, fairness - not the P99
Can We Use Caching to Reduce the P99?

Belief: No

But: latency is not a constant

Caching can reduce P99 request latency!

Effectiveness in Microsoft’s architecture?
Effectiveness Of Caching at Microsoft

Observations for xbox.com (3/2018):

Balance load: steal from cache-rich give to cache-poor

Outsized impact of small reductions in backend load
Our Experimental Prototype

Partition Microsoft’s aggregation cache by backend system

Minimize request P99 by dynamically adjusting partition sizes

Deployable on off-the-shelf software stack

Scalable in #backends, #aggregation servers
Challenges in Minimizing the Request P99

Reallocate from Cache-Rich to Cache-Poor

Definition of “poor”? 1) Cache-Poor = High Query Rate?
Challenges in Minimizing the Request P99

Reallocate from Cache-Rich to Cache-Poor

Definition of “poor”? 2) Cache-Poor = High Query P99?

Query latency insufficient: need to find “cause” of request P99
Basic RobinHood Algorithm

1. Sort all request latencies:

```
P0                  P99                  P100
```

2. Determine who “blocked” P99 request (= on critical path)

```
[Green blocked]
```

3. Allocate cache space to blocking backend

Challenges:
- Not a single cause
- Slow to adapt

Consider a “neighborhood” of the P99
Refined RobinHood Algorithm

1. Sort all request latencies:

2. \( S = \{ \text{requests in P99 neighborhood} \} \)

3. Determine who “blocked” requests in \( S \)

4. Allocate in proportion to “request blocking count” (RBC) in \( S \)

Challenges:
- Not a single cause
- Slow to adapt

Find the backend “causing” high request P99

Consider a “neighborhood” of the P99
Dynamic Reallocation with RobinHood

Record request latencies

Calculate RBC (steps 1 - 3)

Take 1% cache space from every partition. Reallocate in proportion to RBC (step 4)

Record request latencies

Per request:
- latency
- blocking backend

Δ seconds Δ seconds
RobinHood Architecture

Aggregation Cache
- need support for dynamic resizing
- e.g., memcached
  (off-the-shelf version 1.5)

RobinHood Controller
- not on request path
- lightweight python
  - computes RBC
  - runs allocation algorithm
  - controls cache partitioning
RobinHood Architecture

Production system: 16-64 Ag. servers

⇒ RH-control / Ag. server

Distributed RobinHood:

- Local measurements
  - Increase #tail data points
  - Stream to/Pull from central buffer (RH-stats)
  - “Just a buffer” (15s state)

- Local decisions
  - Based on allocation speed
  - Can differ across servers

Constraints on $\Delta$:
- $\Delta = 5$ seconds
- Sufficient # tail data points
- Cache reallocation delay
Experimental Setup

Request generator

Replay production trace
For 4 hours, 200k queries/second (peak: ~500k queries / second)

32 GB cache size
16 threads, 8 Gbit/s network

20 backend clusters
up to 8 servers each

Emulate query latency spikes

Ag. servers

Backends
MySQL (I/O Bound)
Matrix Multiply (CPU Bound)
K-V Store (CPU Bound)
Evaluation Results: P99 Request Latency

RobinHood
[our proposal]

MS Production System
[OneRF]

Maximize Overall Hit Ratio
[Cliffhanger, NSDI’16]

Fairness Between Partitions
[FairRide, NSDI’16]

Balance Query Latencies
[Hyberbolic, ATC’17] (P99 variant)
Evaluation Results: RBC Balance

RBC = request blocking count

Intuition: balanced $\leftrightarrow$ no bottleneck

RobinHood balances RBCs by trading off the performance of low-RBC backends
Conclusions

Is it possible to use caches to improve the request P99?
Yes! SLO violations down to 0.3%, from 30%.
⇒ Use cache as load balancers: “RBC load metric”.

Feasibility in production systems?
Yes! Built using off-the-shelf software stack. Works orthogonally to existing load balancing and auto scaling techniques. In progress.

Is this the optimal solution? End of this project?
No! There’s a lot to do, e.g., other workloads and architectures.

Next steps: performance model → optimality, robustness
(Background in modeling, e.g., Performance’14, Sigmetrics’14,’18)