RIPQ: Advanced Photo Caching on Flash for Facebook

Linpeng Tang (Princeton)
Qi Huang (Cornell & Facebook)
Wyatt Lloyd (USC & Facebook)
Sanjeev Kumar (Facebook)
Kai Li (Princeton)
2 Billion* Photos Shared Daily

* Facebook 2014 Q4 Report
Photo Caches

Close to users
Reduce backbone traffic

Co-located with backend
Reduce backend IO

Photo Serving Stack

Edge Cache

Origin Cache

Storage Backend

Flash
An Analysis of Facebook Photo Caching [Huang et al. SOSP’13]

Advanced caching algorithms help!

Segmented LRU-3: 10% less backbone traffic

Greedy-Dual-Size-Frequency-3: 23% fewer backend IOs

Photo Serving Stack

Edge Cache

Origin Cache

Storage Backend

Flash
In Practice

FIFO was still used

No known way to implement advanced algorithms efficiently

Photo Serving Stack

Edge Cache

Origin Cache

Storage Backend

Flash
Advanced caching helps:

- 23% fewer backend IOs
- 10% less backbone traffic

Difficult to implement on flash:

- FIFO still used

Restricted Insertion Priority Queue:

efficiently implement advanced caching algorithms on flash
Outline

• Why are advanced caching algorithms difficult to implement on flash efficiently?

• How RIPQ solves this problem?
  – Why use priority queue?
  – How to efficiently implement one on flash?

• Evaluation
  – 10% less backbone traffic
  – 23% fewer backend IOs
Outline

• Why are advanced caching algorithms difficult to implement on flash efficiently?
  – Write pattern of FIFO and LRU

• How RIPQ solves this problem?
  – Why use priority queue?
  – How to efficiently implement one on flash?

• Evaluation
  – 10% less backbone traffic
  – 23% fewer backend IOs
FIFO Does Sequential Writes

Cache space of FIFO

Head  Tail
FIFO Does Sequential Writes

Cache space of FIFO

Head Miss Tail
FIFO Does Sequential Writes

Cache space of FIFO

Head → Hit → Tail
FIFO Does Sequential Writes

Cache space of FIFO

Head \[\text{Evicted}\] Tail

No random writes needed for FIFO
LRU Needs Random Writes

Locations on flash ≠ Locations in LRU queue
LRU Needs Random Writes

Cache space of LRU

Non-contiguous on flash

Random writes needed to reuse space
Why Care About Random Writes?

• Write-heavy workload
  – Long tail access pattern, moderate hit ratio
  – Each miss triggers a write to cache

• Small random writes are harmful for flash
  – e.g. Min et al. FAST’12
  – High write amplification
    - Low write throughput
    - Short device lifetime
What write size do we need?

- **Large writes**
  - High write throughput at high utilization
  - 16~32MiB in Min et al. FAST’2012

- **What’s the trend since then?**
  - Random writes tested for 3 modern devices
  - 128~512MiB needed now

100MiB+ writes needed for efficiency
Outline

• Why are advanced caching algorithms difficult to implement on flash efficiently?

• How RIPQ solves this problem?

• Evaluation
RIPQ Architecture
(Restricted Insertion Priority Queue)

Advanced Caching Policy
(SLRU, GDSF ...)

Priority Queue API

Approximate Priority Queue

Flash-friendly Workloads

RAM

Flash

Caching algorithms approximated as well

Efficient caching on flash
RIPQ Architecture
(Restricted Insertion Priority Queue)

Advanced Caching Policy
(SLRU, GDSF ...)

Priority Queue API

Approximate Priority Queue

Flash-friendly Workloads

RAM

Flash

Restricted insertion
Section merge/split
Large writes
Lazy updates
Priority Queue API

- No single best caching policy

- **Segmented LRU** [Karedla’94]
  - Reduce both backend IO and backbone traffic
  - SLRU-3: best algorithm for Edge so far

- **Greedy-Dual-Size-Frequency** [Cherkasova’98]
  - Favor small objects
  - Further reduces backend IO
  - GDSF-3: best algorithm for Origin so far
Segmented LRU

- Concatenation of K LRU caches

Cache space of SLRU-3

Head → L3 → L2 → L1 → Tail

Miss
Segmented LRU

• Concatenation of K LRU caches

Cache space of SLRU-3

Head  L3  L2  L1  Tail

Miss
Segmented LRU

- Concatenation of K LRU caches

Cache space of SLRU-3

Head L3 L2 L1 Tail

Hit
Segmented LRU

- Concatenation of K LRU caches

Cache space of SLRU-3

Head  L3  L2  L1  Tail

Hit  again
Greedy-Dual-Size-Frequency

- Favoring small objects
Greedy-Dual-Size-Frequency

- Favoring small objects

Cache space of GDSF-3

Head — Miss — Tail
Greedy-Dual-Size-Frequency

• Favoring small objects

Cache space of GDSF-3

Miss
Greedy-Dual-Size-Frequency

- Favoring small objects

Cache space of GDSF-3

- Write workload more random than LRU
- Operations similar to priority queue
Relative Priority Queue for Advanced Caching Algorithms

Miss object: insert(x, p)
Relative Priority Queue for Advanced Caching Algorithms

Head \[ 1.0 \quad \rho' \quad \text{Cache space} \quad 0.0 \quad \text{Tail} \]

Hit object: \( \text{increase}(x, \rho') \)
Relative Priority Queue for Advanced Caching Algorithms

Implicit demotion on insert/increase:

- Object with lower priorities moves towards the tail
Relative Priority Queue for Advanced Caching Algorithms

Relative priority queue captures the dynamics of many caching algorithms!
RIPQ Design: Large Writes

- Need to buffer objects (10s KiB) into block writes
- Once written, blocks are immutable!
- 256MiB block size, 90% utilization
  - Large caching capacity
  - High write throughput
RIPQ Design: Restricted Insertion Points

- Exact priority queue
  - Insert to any block in the queue
- Each block needs a separate buffer
  - Whole flash space buffered in RAM!
RIPQ Design: Restricted Insertion Points

Solution: restricted insertion points
Section is Unit for Insertion

1 .. 0.6 0.6 .. 0.35 0.35 .. 0

Head

Section

Section

Section

Tail

Active block with RAM buffer

Sealed block on flash

Each section has one insertion point
Section is Unit for Insertion

Insert procedure
• Find corresponding section
• Copy data into active block
• Updating section priority range

insert(x, 0.55)
Section is Unit for Insertion

Relative orders within one section not guaranteed!
Trade-off in Section Size

Section size controls approximation error

- Sections, approximation error
- Sections, RAM buffer
RIPQ Design: Lazy Update

Naïve approach: copy to the corresponding active block

Problem with naïve approach
- Data copying/duplication on flash
Solution: use **virtual block** to track the updated location!
RIPQ Design: Lazy Update

Solution: use virtual block to track the updated location!
Virtual Block Remembers Update Location

No data written during virtual update
Actual Update During Eviction

Section

Head

Section

Section

x now at tail block.
Actual Update During Eviction

Copy data to the active block

Always one copy of data on flash
RIPQ Design

• Relative priority queue API

• RIPQ design points
  – Large writes
  – Restricted insertion points
  – Lazy update
  – Section merge/split
    • Balance section sizes and RAM buffer usage

• Static caching
  – Photos are static
Outline

• Why are advanced caching algorithms difficult to implement on flash efficiently?

• How RIPQ solves this problem?

• Evaluation
Evaluation Questions

• How much RAM buffer needed?

• How good is RIPQ’s approximation?

• What’s the throughput of RIPQ?
Evaluation Approach

• Real-world Facebook workloads
  – Origin
  – Edge

• 670 GiB flash card
  – 256MiB block size
  – 90% utilization

• Baselines
  – FIFO
  – SIPQ: Single Insertion Priority Queue
RIPQ Needs Small Number of Insertion Points

- **Exact GDSF-3**
- **Exact SLRU-3**
- **FIFO**

- Gain of +16% for Exact GDSF-3
- Gain of +6% for Exact SLRU-3

Object-wise hit-ratio (%)

Insertion points
RIPQ Needs Small Number of Insertion Points

![Graph showing hit-ratios for different schemes with varying insertion points. The graph plots object-wise hit-ratios (%) against insertion points. The schemes compared include Exact GDSF-3, Exact SLRU-3, SLRU-3, and FIFO.]
RIPQ Needs Small Number of Insertion Points

You don’t need much RAM buffer (2GiB)!
RIPQ Has High Fidelity

Object-wise hit-ratio (%)

SLRU-1  SLRU-2  SLRU-3  GDSF-1  GDSF-2  GDSF-3  FIFO
RIPQ Has High Fidelity
RIPQ Has High Fidelity

RIPQ achieves \( \leq 0.5\% \) difference for all algorithms
RIPQ Has High Fidelity

+16% hit-ratio  ➔  23% fewer backend IOs
RIPQ Has High Throughput

RIPQ throughput comparable to FIFO (≤10% diff.)
Related Works

RAM-based advanced caching
SLRU(Karedla’94), GDSF(Young’94, Cao’97, Cherkasova’01), SIZE(Abrams’96), LFU(Maffeis’93), LIRS (Jiang’02), ...

RIPQ enables their use on flash

Flash-based caching solutions
Facebook FlashCache, Janus(Albrecht ’13), Nitro(Li’13), OP-FCL(Oh’12), FlashTier(Saxena’12), Hec(Yang’13), ...

RIPQ supports advanced algorithms

Flash performance
Stoica’09, Chen’09, Bouganim’09, Min’12, ...

Trend continues for modern flash cards
RIPQ

• First framework for advanced caching on flash
  – Relative priority queue interface
  – Large writes
  – Restricted insertion points
  – Lazy update
  – Section merge/split

• Enables SLRU-3 & GDSF-3 for Facebook photos
  – 10% less backbone traffic
  – 23% fewer backend IOs