Concurrent Caching @ Google

Charles Fry (fry@google.com)
Ben Manes (bmanes@google.com)

Introduction

Evolution of in-memory caching in Java

- API evolution
- Implementation evolution

Rationale for the work

- Earlier performance was deficient for real workloads
  - Improved concurrency
  - Throughput
  - Response times
- Need for a canonical, Google-wide caching library
  - Fast, flexible, and feature rich
  - It's hard to roll your own!

API Evolution

LinkedHashMap
- Separate implementation for every combination of features (e.g. eviction, expiration, refresh)

ConcurrentLinkedHashMap
- Concurrent LRU cache, supporting weighted values

ReferenceMap
- Concurrent map supporting weak/soft keys & values

ReferenceCache
- Self-populating ReferenceMap

MapMaker
- Introduced builder API

CacheBuilder
- Introduced Cache interface

Map as a Cache

MapMaker was based on the paradigm of exposing a cache as a self-populating (computing) map

- Invoking get on the map resulted in the creation of a new cached entry
- Calling size on the map returned the number of cached elements

Map as a Cache

Disadvantages of this approach:

- Either pending computations can be overwritten by explicit writes or writes must block on pending computations
- Map.get causes a type-safety hole
- Map.get is a read operation and users don't expect it to also write
- Map.equals is not symmetric on a self-populating Map
- No way to lookup without causing computation
- Interface fails to convey the intent (caching!)

Cache Interface

Cache resembles Map superficially, but adds caching-specific methods:

```
// created using CacheBuilder
interface Cache {
  V get(K key) throws ExecutionException;
  void invalidate(Object key);
  int size();
  ConcurrentMap<K, V> asMap();
  CacheStats stats();
  void cleanUp();
}
```

CacheBuilder Feature Overview

Adjustable write concurrency (segmented hashtable)
Full read concurrency
Self populating
Notifies on removal
Soft/weak references
Bounds by a maximum size
Expires based on usage policies

Future?

- Batch computations
- LIRS eviction policy
- Weighted entries
- Auto-refresh

Implementation Evolution

All eviction types introduce the need for frequent writes:

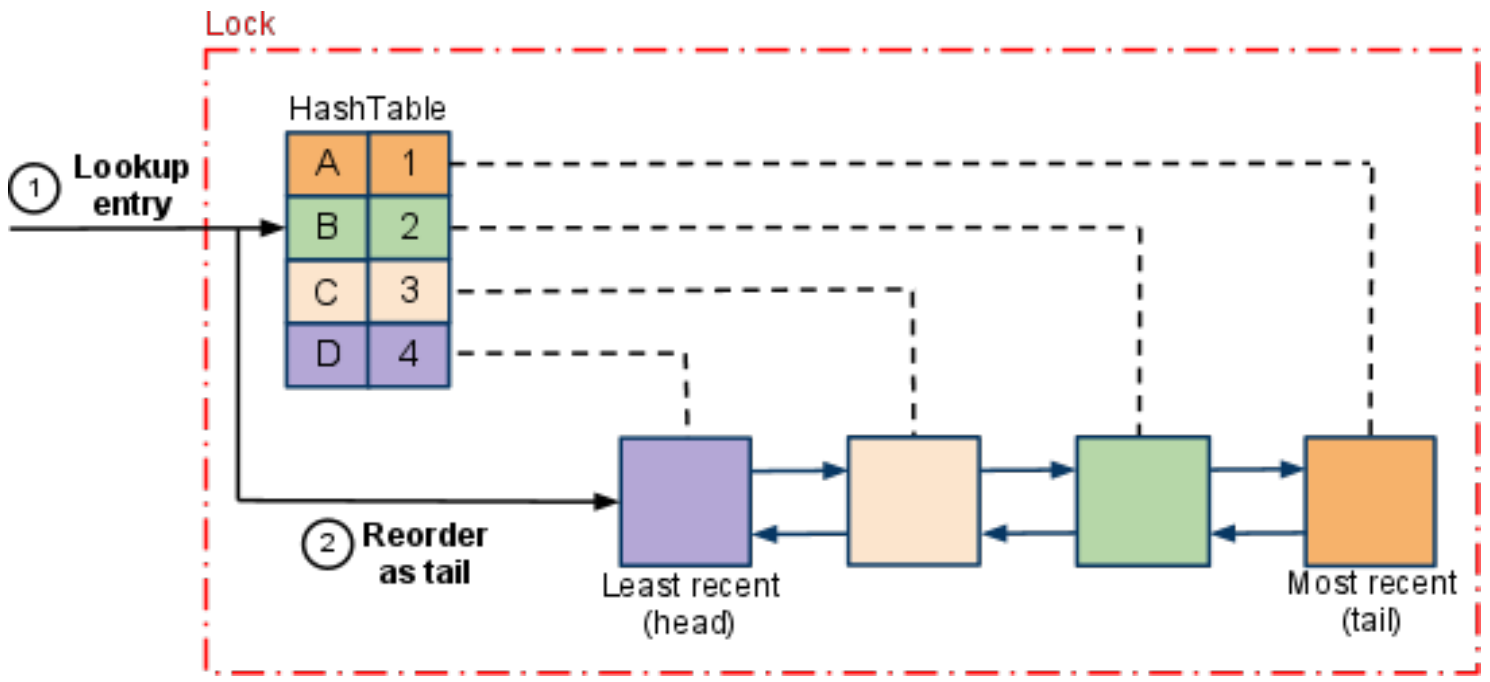
- Size-based eviction must record the order of each access
- Timed expiration must...
  - Record the time and order of each access
  - Cleanup stale entries which have expired
- Reference cleanup must cleanup stale entries whose key or value has been garbage collected

Traditional implementations require locking on every read to maintain the history and perform cleanup tasks

Traditional: Size-based Eviction

Commonly implemented with LinkedHashMap

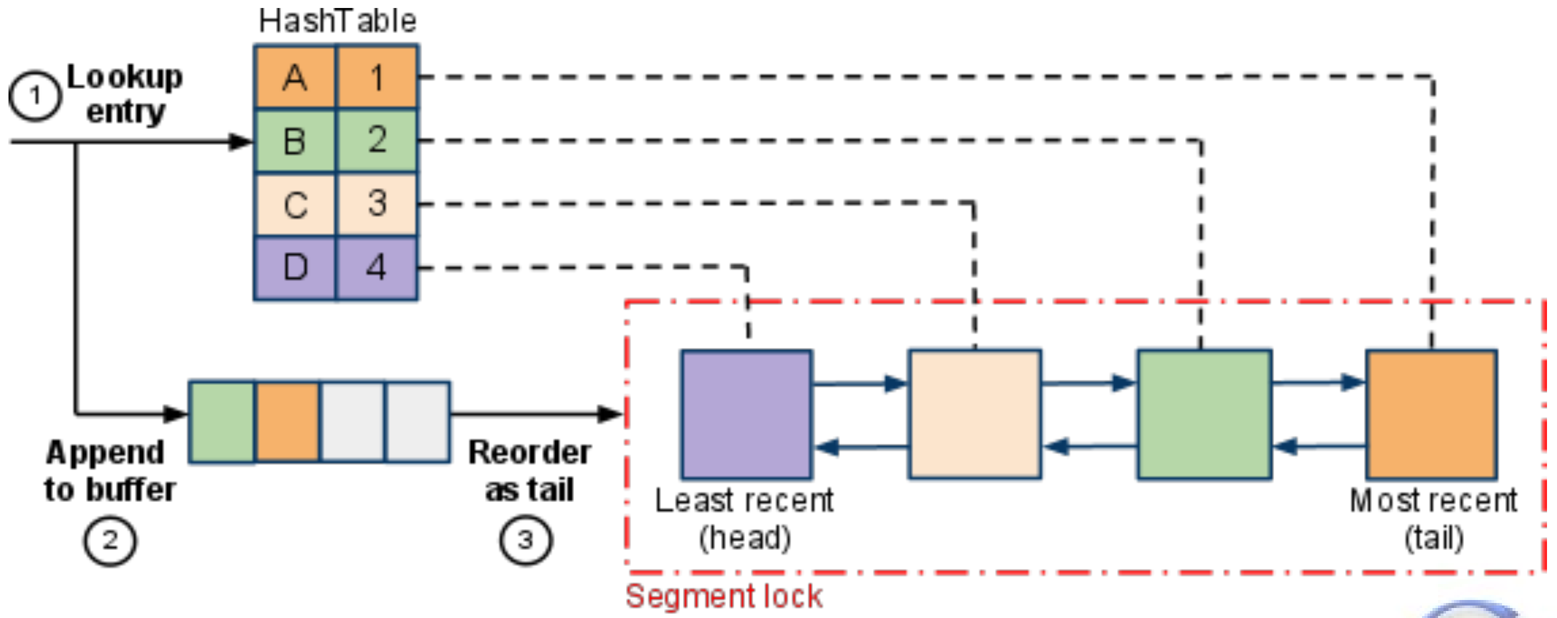
- Not concurrent (for either reads or writes)
- Can evict based on access or insertion order
- Access can result in structural modifications



CacheBuilder: Size-based Eviction

Each segment maintains an access-ordered (LRU) list

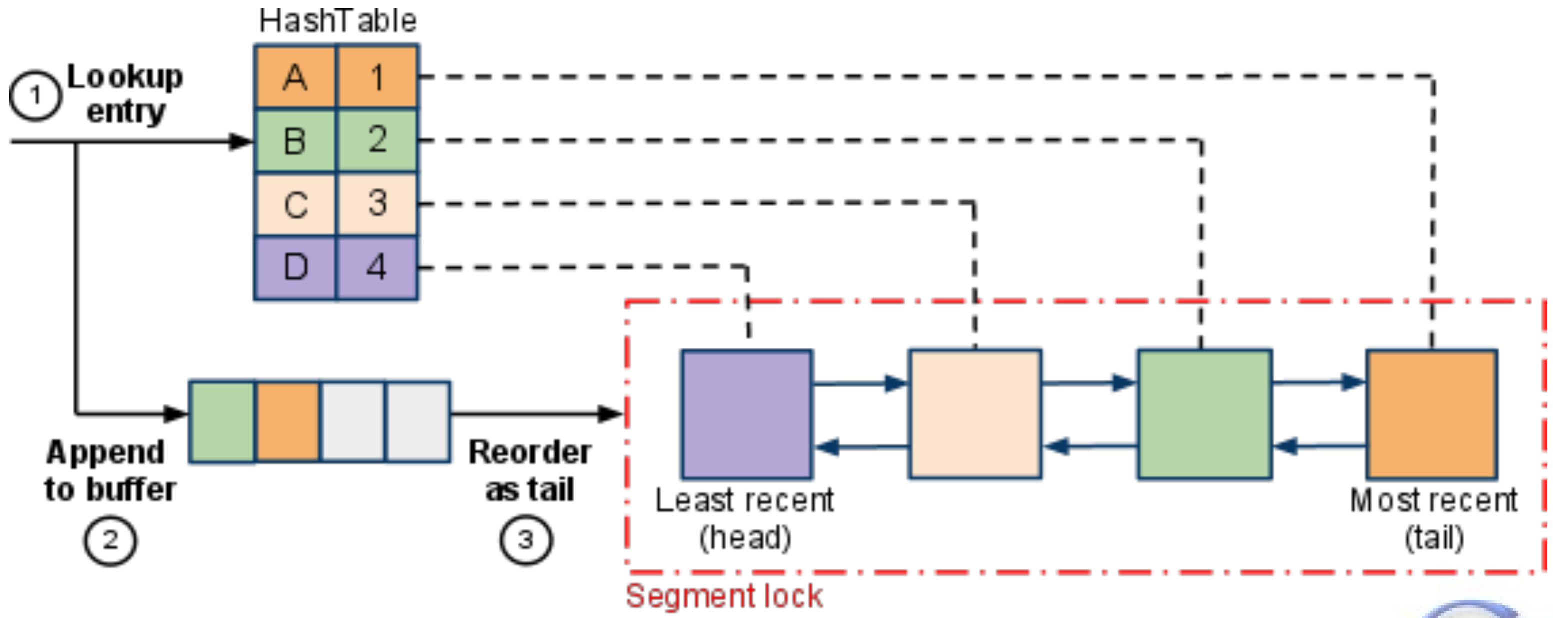
- LRU modifications guarded by the segment lock
- Implemented by linking internal hash entries
- Only approximates strict LRU



CacheBuilder: Size-based Eviction

A memento of a read operation is enqueued in the buffer
The read buffer is drained under lock...

- On every write (already holds the lock)
- After a threshold buffer size is reached (tryLock)



Traditional: Timed Expiration

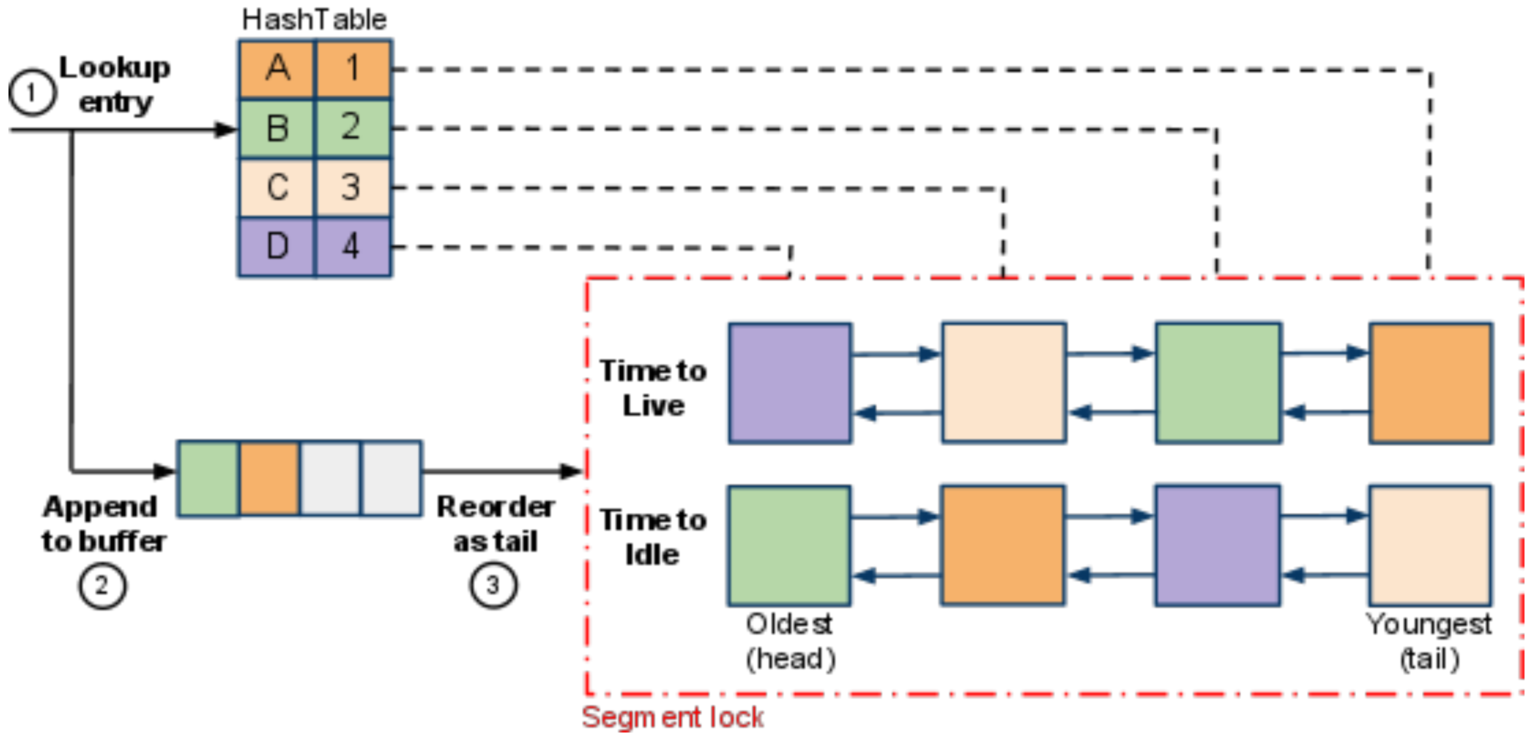
Commonly implemented using a Timer thread

- Expiration events added to a priority queue, O(log n)
- Expiration events are weak references
- Events remain until timer has expired
    - Some events may be stale by the time they're polled

CacheBuilder: Timed Expiration

Expiration can be specified as

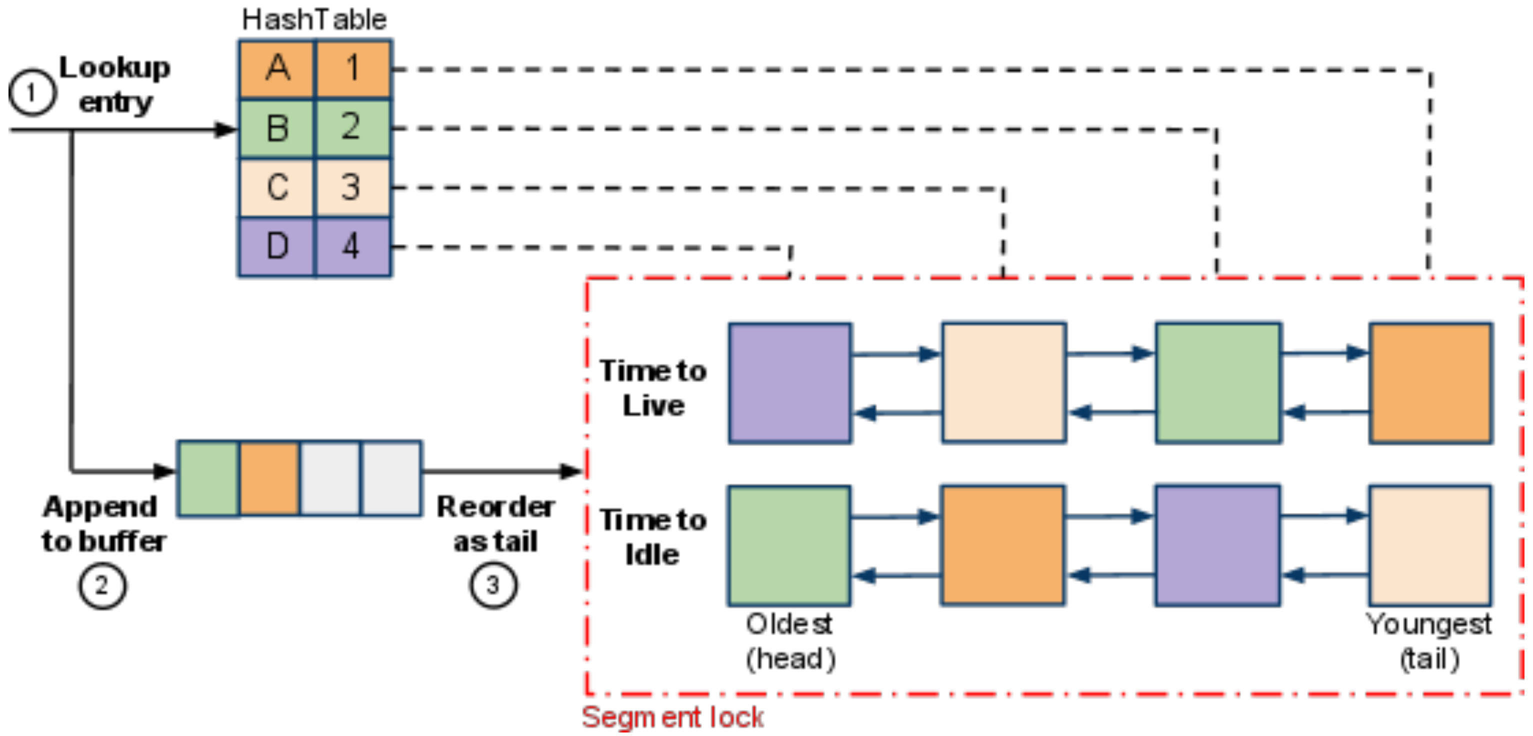
- Time-to-live (expireAfterWrite)
- Time-to-idle (expireAfterAccess)
- Can be periodically swept by an auxiliary thread



CacheBuilder: Timed Expiration

Implemented as a time-based LRU list

- Hash entries include an expiration time
- Head of the list is the next element to be expired
- Read/write access moves elements to the tail
- Implemented by linking internal hash entries
- Concurrency achieved by reusing buffers from size-based eviction



Traditional: Reference Cleanup

Reference eviction can be specified as

- Weak keys
- Weak or soft values
- Hash entry removed when key or value is collected

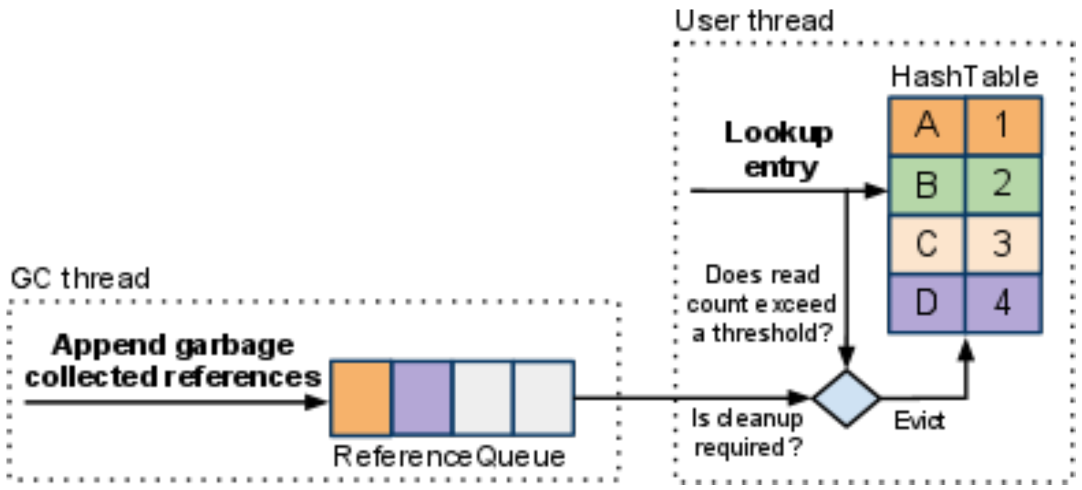
FinalizableReferenceQueue

- Globally shared ReferenceQueue
- Entries have a cleanup method for removal
- Each removal acquires the lock
- Single thread performs cleanup
- Cleanup thread may fall behind under production load
- Thread was incompatible with AppEngine and J2EE

CacheBuilder: Reference Cleanup

Each cache manages its own ReferenceQueues

- Queue draining is batched under segment lock
- Amortize draining on user operations
  - Drains on all writes and periodically on reads
  - Can be periodically drained by an auxiliary thread



History and Cleanup

The frequent writes required by each eviction strategy can be performed best-effort, within certain bounds

- History management is delayed until the next eviction or the buffer exceeds a threshold size
- Cleanup doesn't have any hard timing guarantees

These writes can be batched under the segment lock and executed...

- Immediately on user writes
- Within a tryLock on user reads
- Within a tryLock on a cleanup thread

Conclusion

Future

- Continue to add features to CacheBuilder
- Mature algorithms and optimize

JSR-166

- Working with Doug Lea on a standard library
- New ConcurrentHashMap implementation in JDK8!
    - No longer segmented hash-table

Guava

Google's Core Libraries for Java!

- Download it, see online javadocs, etc.
  - http://guava-libraries.googlecode.com
- Join discussion list
  - http://groups.google.com/group/guava-discuss
- Ask for help
  - Post with `guava` tag to StackOverflow.com

Q&A

FIN