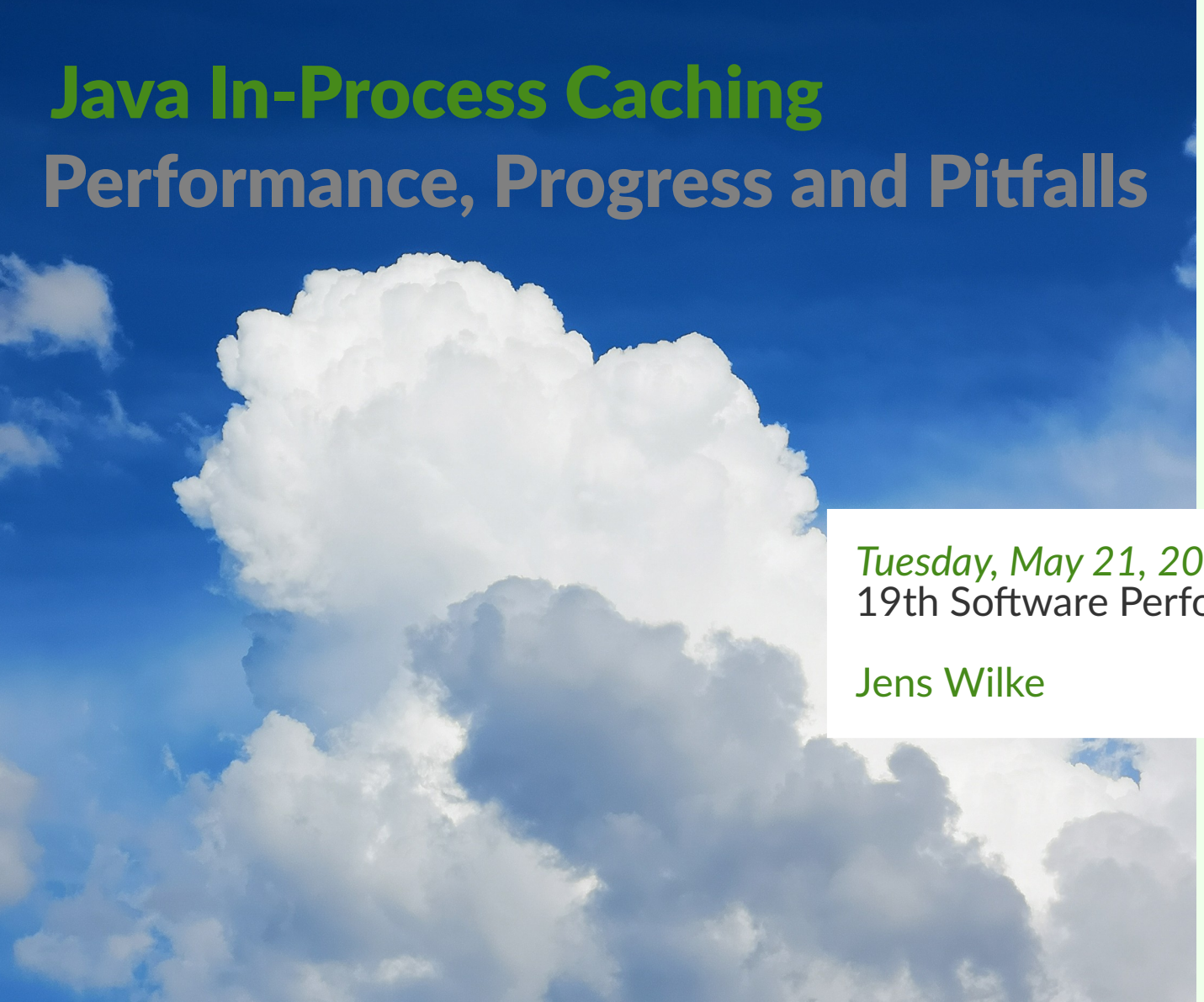


Java In-Process Caching
Performance, Progress and Pitfalls

Tuesday, May 21, 2019
19th Software Performance Meetup, Munich

Jens Wilke

Links - Disclaimer - Copyright

talk slides and diagram source data

https://github.com/cruftex/talk-java-in-process-caching-performance-progress-pitfalls

used benchmarks

https://github.com/cache2k/cache2k-benchmark

disclaimer

No guarantees at all. Do not sue me for any mistake, instead, send a pull request and correct it!

Copyright: Creative Commons Attribution
CC BY 4.0

About Me

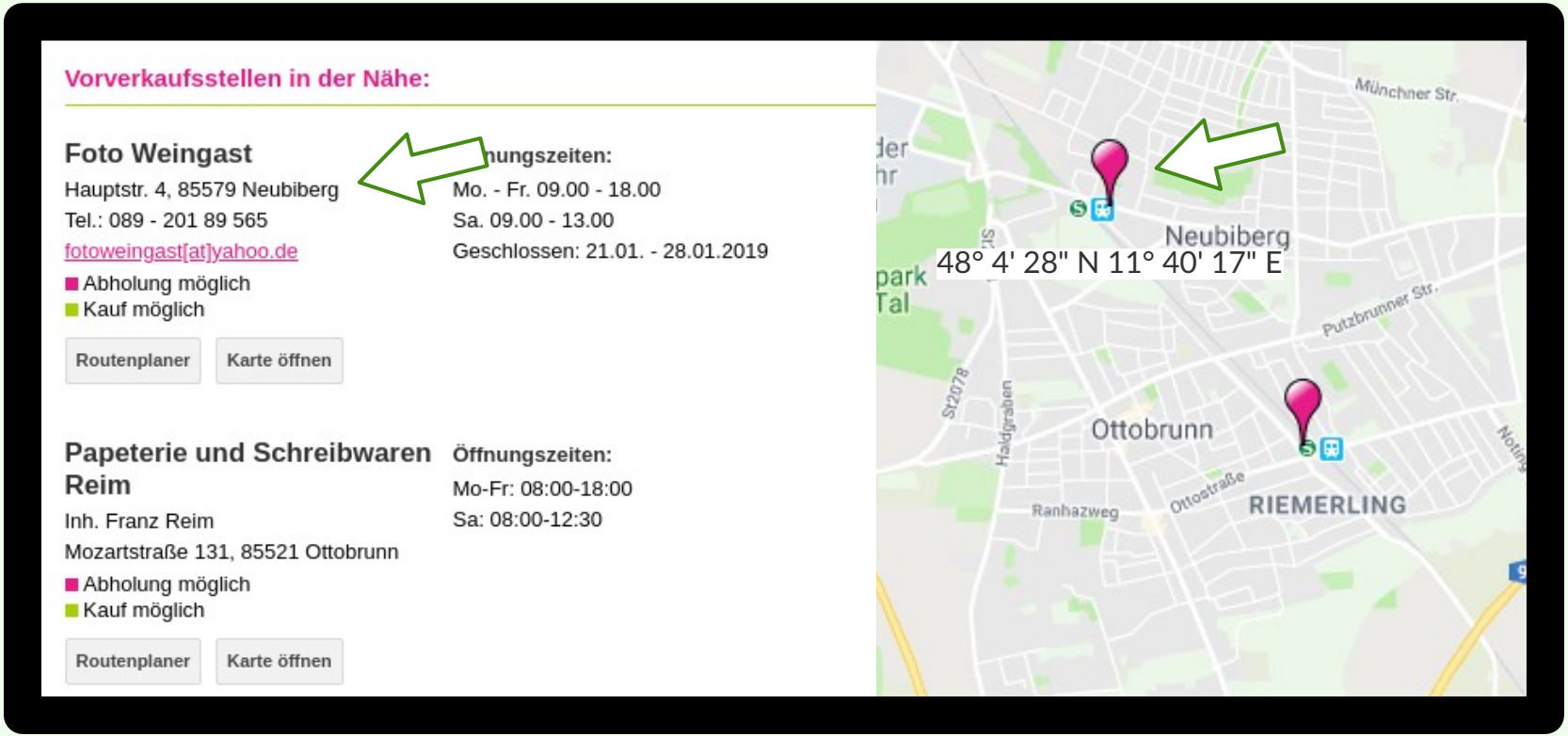
Jens Wilke
@cruftex
cruftex.net

- Performance Fan(atic)
- Java Hacker since 1998
- Author of cache2k
- 70+ answered questions on StackOverflow about Caching
- JCache / JSR107 Contributor

Up Next

Java In-Process Caching What and Why

Example 1: Geolocation Lookup



Example 2: Date Formatting

SPONTAN DABEI SEIN

	**Joe Bonamassa** MO 20.05.2019, 20:00 Uhr	Olympiapark München, Olympiahalle	TICKETS SICHERN
	**Thad Jones Big Band - Thomas Gansch** MO 20.05.2019, 20:00 Uhr	Freiheizhalle	TICKETS SICHERN
	**Elephant** MO 20.05.2019, 20:00 Uhr	Münchner Kammerspiele	TICKETS SICHERN
	**Brutus** MO 20.05.2019, 21:00 Uhr	STROM	TICKETS SICHERN
	**Kino am Olympiasee - Der OpenAir Kinosommer im Olympiapark - Der Junge muss an die frische Luft** MO 20.05.2019, 21:00 Uhr	Olympiapark München, Freifläche der Olympia-Schwimmhalle	TICKETS SICHERN

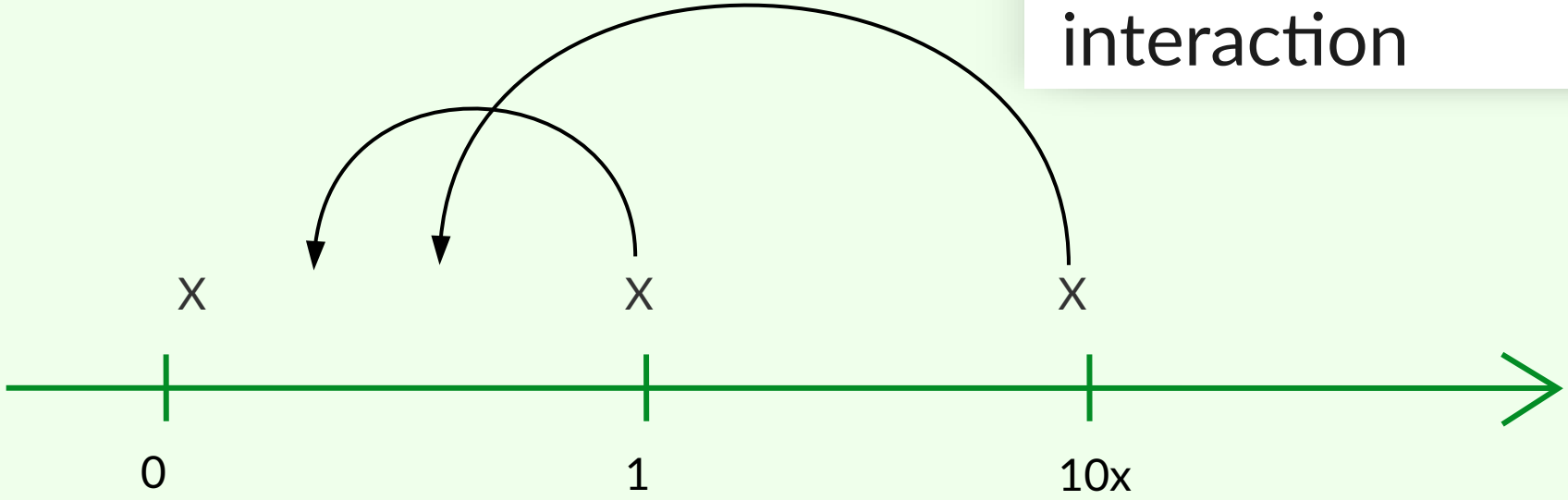
Expensive Operations per Web Request

How often is an operation executed per web request or user interaction?

- Less than once:
  e.g. initialization on startup
- Exactly once:
  e.g. fetch data and resolve geolocation
- More than once:
  e.g. render a time or date

Reduce Expensive Operations

Cache:
Less executions per web request or user interaction



(Java) Caching

technical

- temporary data storage to serve requests faster
- reduce expensive operations at the cost of storage

benefits

- A tool to tune the space time tradeoff problem
- Lower latency and improve UX
- If not because of great UX, let‘s save computing costs!

Java *In Process* Caching

technical

- temporary data storage to serve requests faster
- reduce expensive operations at the cost of *heap memory*
- *keep data as close to the CPU as possible*

benefits

- A tool to tune the space time tradeoff problem
- Lower latency *much more* and improve UX
- If not because of great UX, let‘s save more computing costs!

Constructing an Java In Process Cache

interface

The interface of a cache is similar (sometimes identical) to a Java Map:

```
cache.put(key, value);

value = cache.get(key);
```

implementation

- A hash table
- An eviction strategy
  - to limit the used memory
  - but keep data that is „hot“

Up Next

Benchmark a Simple Cache

Read Only JMH Benchmark

```
@Param({"100000"})
public int entryCount = 100 * 1000;
BenchmarkCache<Integer, Integer> cache;
Integer[] ints;

@Setup
public void setup() throws Exception {
  cache = getFactory().create(entryCount);
  ints = new Integer[PATTERN_COUNT];
  RandomGenerator generator =
    new XorShift1024StarRandomGenerator(1802);
  for (int i = 0; i < PATTERN_COUNT; i++) {
    ints[i] = generator.nextInt(entryCount);
  }
  for (int i = 0; i < entryCount; i++) {
    cache.put(i, i);
  }
}

@Benchmark @BenchmarkMode(Mode.Throughput)
public long read(ThreadState threadState) {
  int idx = (int) (threadState.index++ % PATTERN_COUNT);
  return cache.get(ints[idx]);
}
```

1

2

3

4

Benchmark that does only read a cache that is filled with data intially. No eviction takes place, so we can compare the read throughput with a (concurrent) hash table.

1) create a cache, via a wrapper to adapt to different implementations
2) create an array with random integer objects. Value range does not exceed entry count
3) fill cache once, not part of the benchmark
4) benchmark operation does one cache read with random key

Benchmark Specifications

hardware

- CPU: Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz
  4 physical cores
- Benchmarks are done with different number of cores by Linux CPU hotplugging
- Oracle JVM 1.8.0-131, JMH 1.18
- Ubuntu 14.04, 4.4.0-137-generic

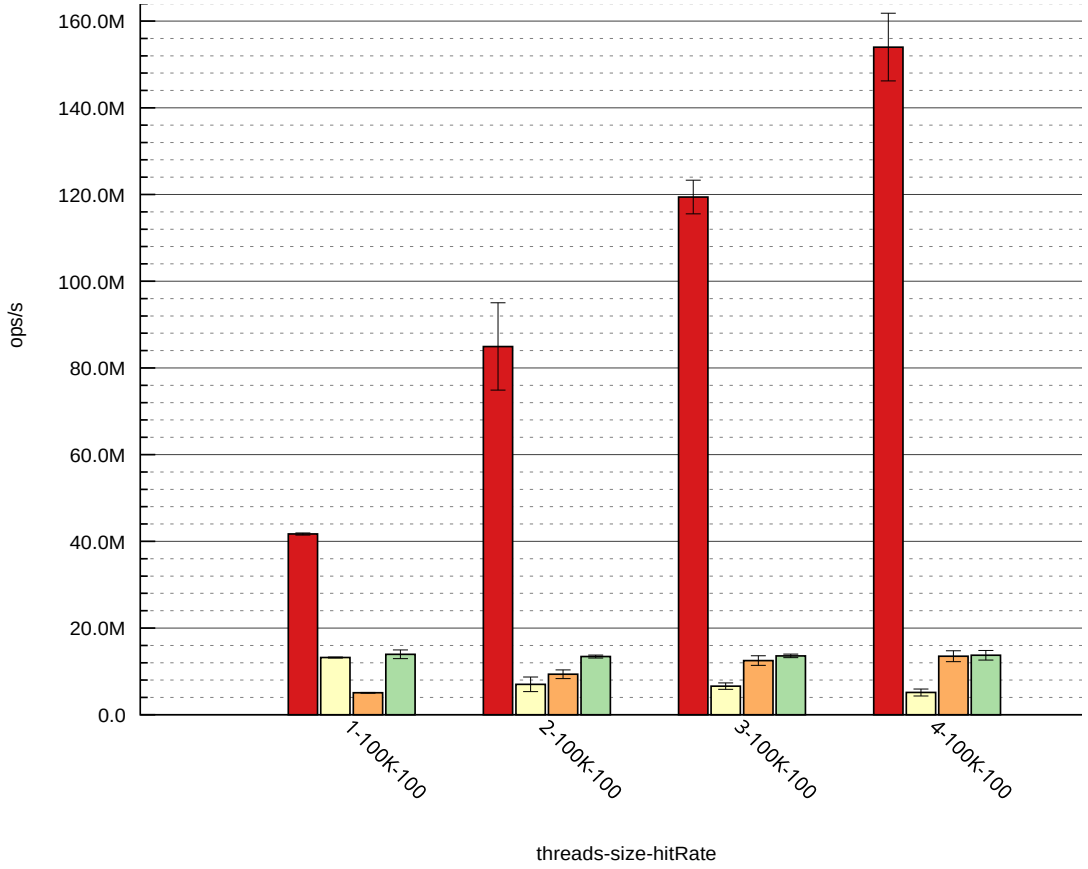
cache versions

- Google Guava Cache, Version 26
- Caffeine, Version 2.6.2
- cache2k, Version 1.2.0.Final
- EHCache, Version 3.6.1

JMH parameters

- 2 forks, 2 warmup iterations, 3 measurement iterations, 15 second iterations times
- => 6 measurement iterations

Results



- ConcurrentHashMap
- Simple Java Implementation with LRU via LinkedHashMap
- Google Guava Cache
- Simple Java Implementation with LRU via LinkedHashMap and Segmentation

Y axis: operations/s

X axis: Number of threads

Especially when multi-threaded the ConcurrentHashMap is much faster then a cache.

Up Next

LRU = Least Recently Used

LRU List Operations

double linked list with three entries

cache operation		list operation
1. put (1, x)		insert new head
2. put (2, x)		insert new head
3. get (1)		move to front
4. put (3, x)		insert new head
5. put (4,x)		remove tail (key 2) insert new head remove tail

LRU Properties

cool…

- Simple and smart algorithm for eviction (or replacement)
- Everybody knows it from CS, „eviction = LRU“

…but:

- List operations need synchronization
- A cache read means rewriting references in 4 objects, most likely touching 4 different CPU cache lines
- A read operation (happens often!) is more expensive than an eviction (happens not so often!)
- LRU is not scan resistent; scans wipe out the working set in the cache
- Non frequently accessed objects need a long time until evicted

LRU Alternatives?

we look for

- Reduce CPU cycles for the read operation
- Do more costly operations later when we need to evict
- Also take frequency into account, keeping more frequently accessed objects longer

lots of research

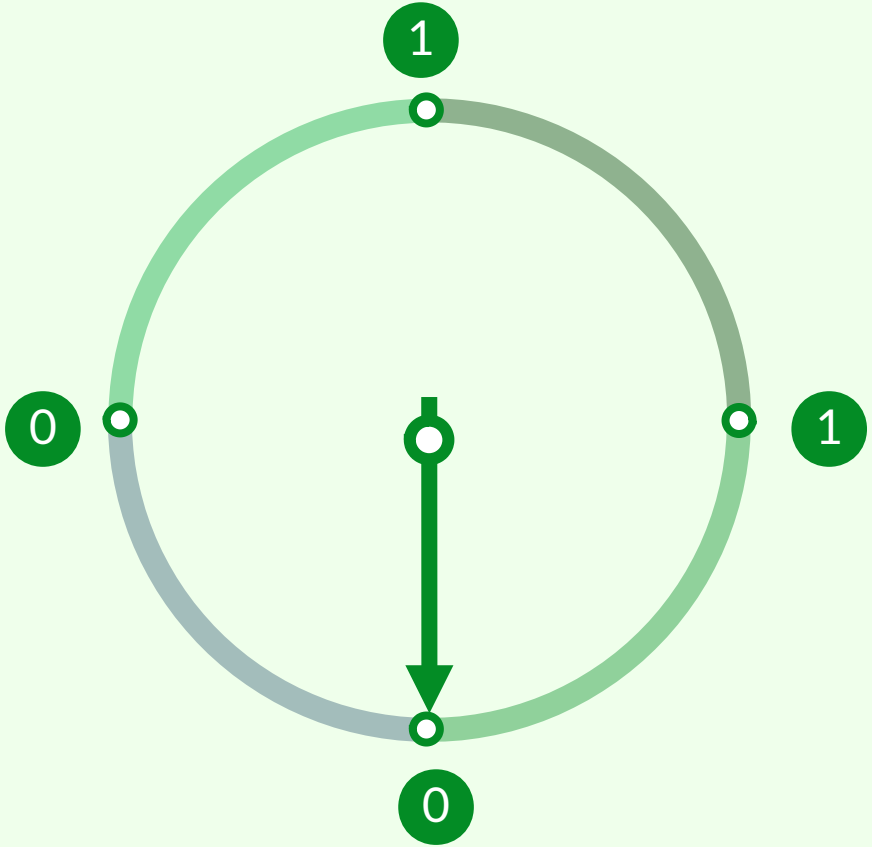
Overview at:

Wikipedia:Page_replacement_algorithm

Up Next

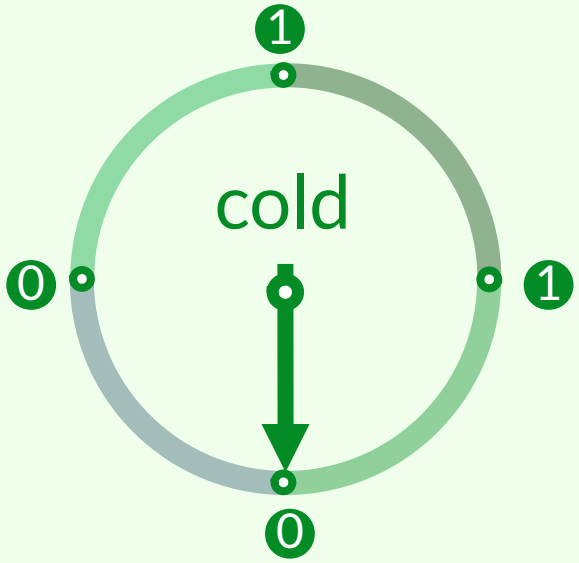
Clock / Clock-Pro Eviction

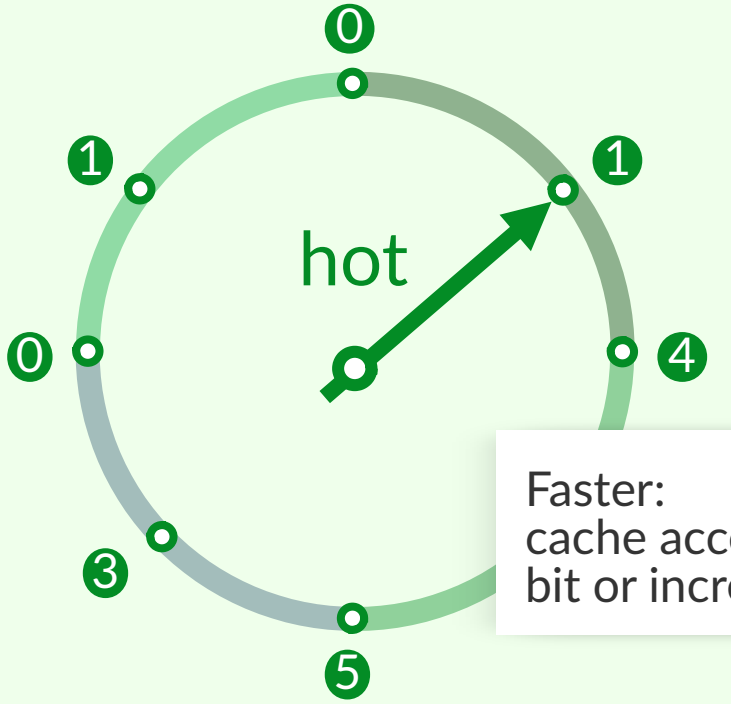
Clock



- Each cache entry has a reference bit which indicates whether the entry was accessed
- Access: Sets reference bit
- Eviction, scan at clock hand:
  - Not-Referenced? Evict!
  - Referenced? Clear reference and move to the next

Clock-Pro





history

Faster:
cache access is tracked by setting a bit or incrementing a counter

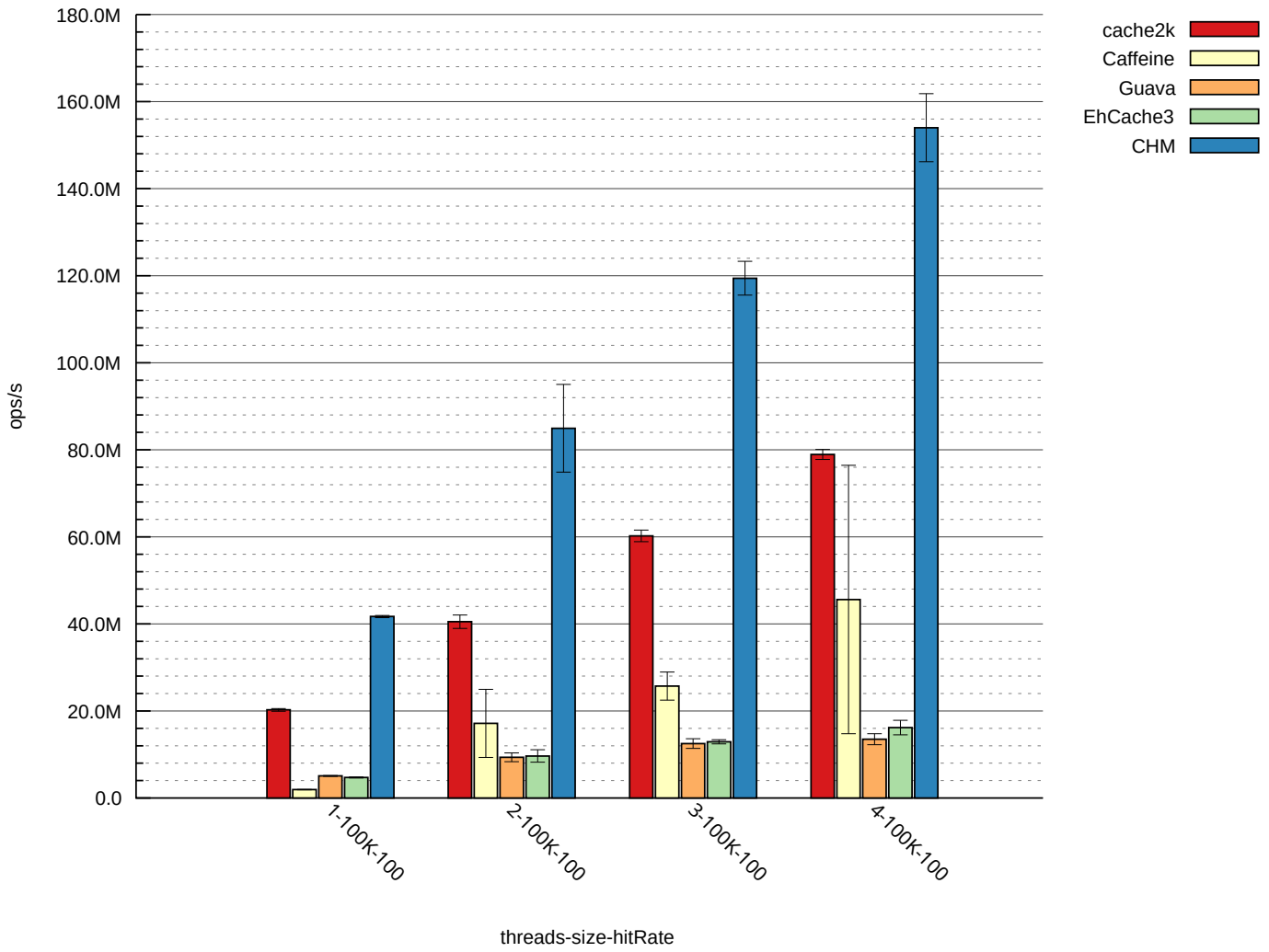
- Extra clock for hot data
- History of recently evicted keys
- cache2k: Use reference counter instead of reference bit

Up Next

Will it Blend?
(more Benchmarks...)

Results

- Google Guava Cache and EHCache3 are slow in comparison to the `ConcurrentHashMap`
- Caffeine is faster, if there are sufficient CPU cores/threads
- Cache2k is fastest, at about half the speed of the `ConcurrentHashMap`



Up Next

What about eviction efficiency?

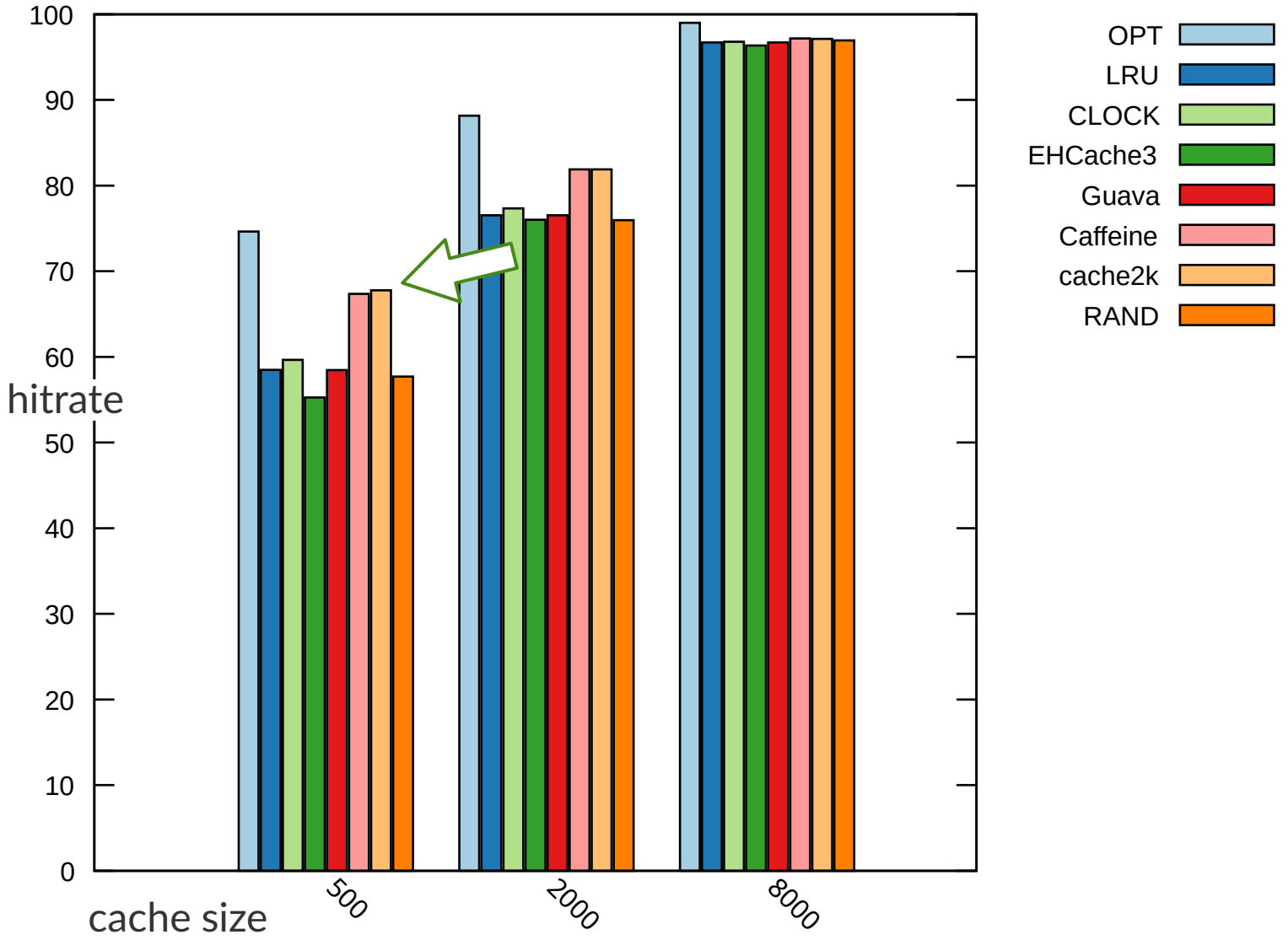
Benchmarking Eviction Quality

- Collect access sequences (traces)
- Replay the access sequence on a cache and count hits and misses

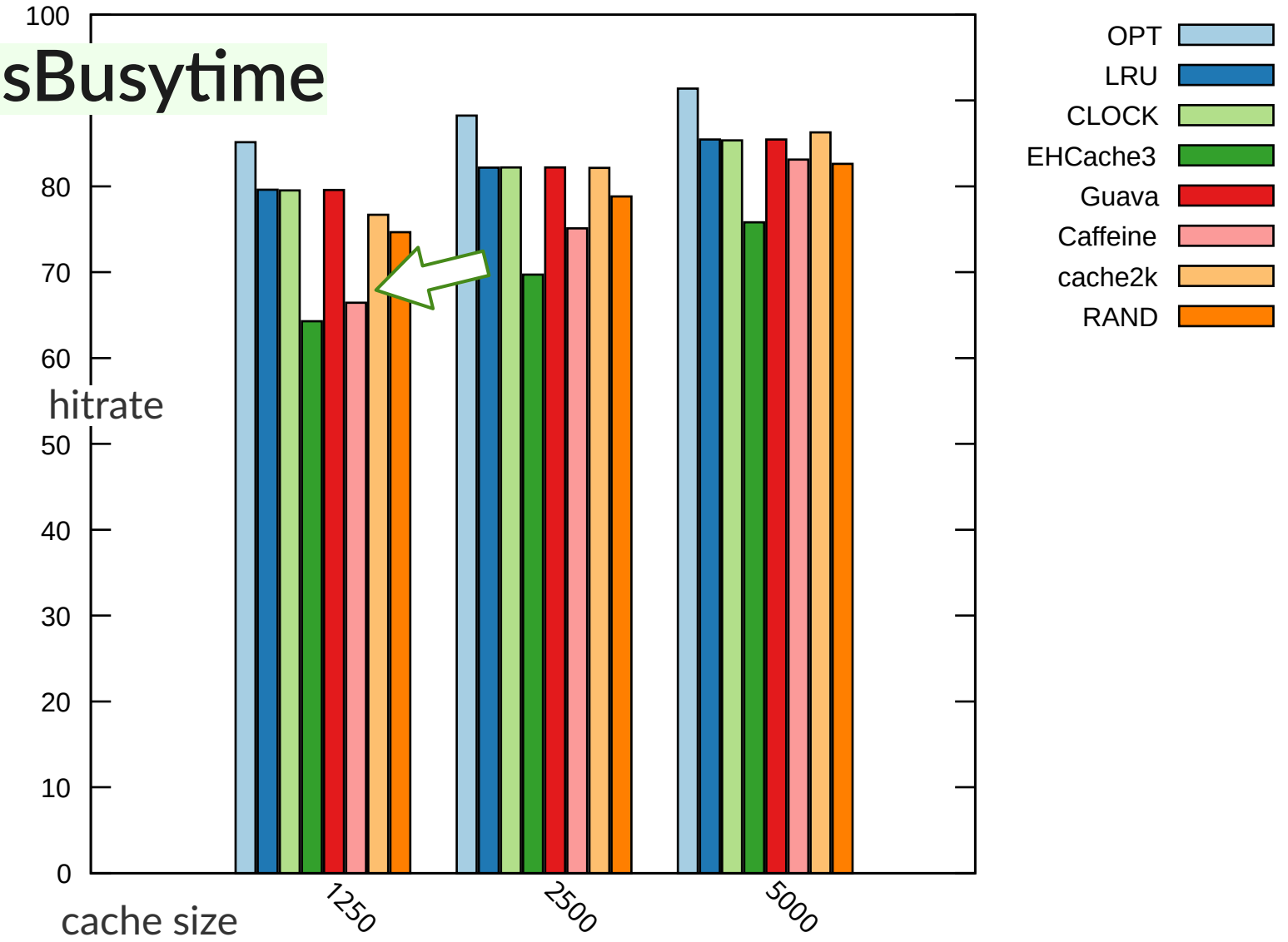
More information about the traces in the blog article:

https://cruftex.net/2016/05/09/Java-Caching-Benchmarks-2016-Part-2.html

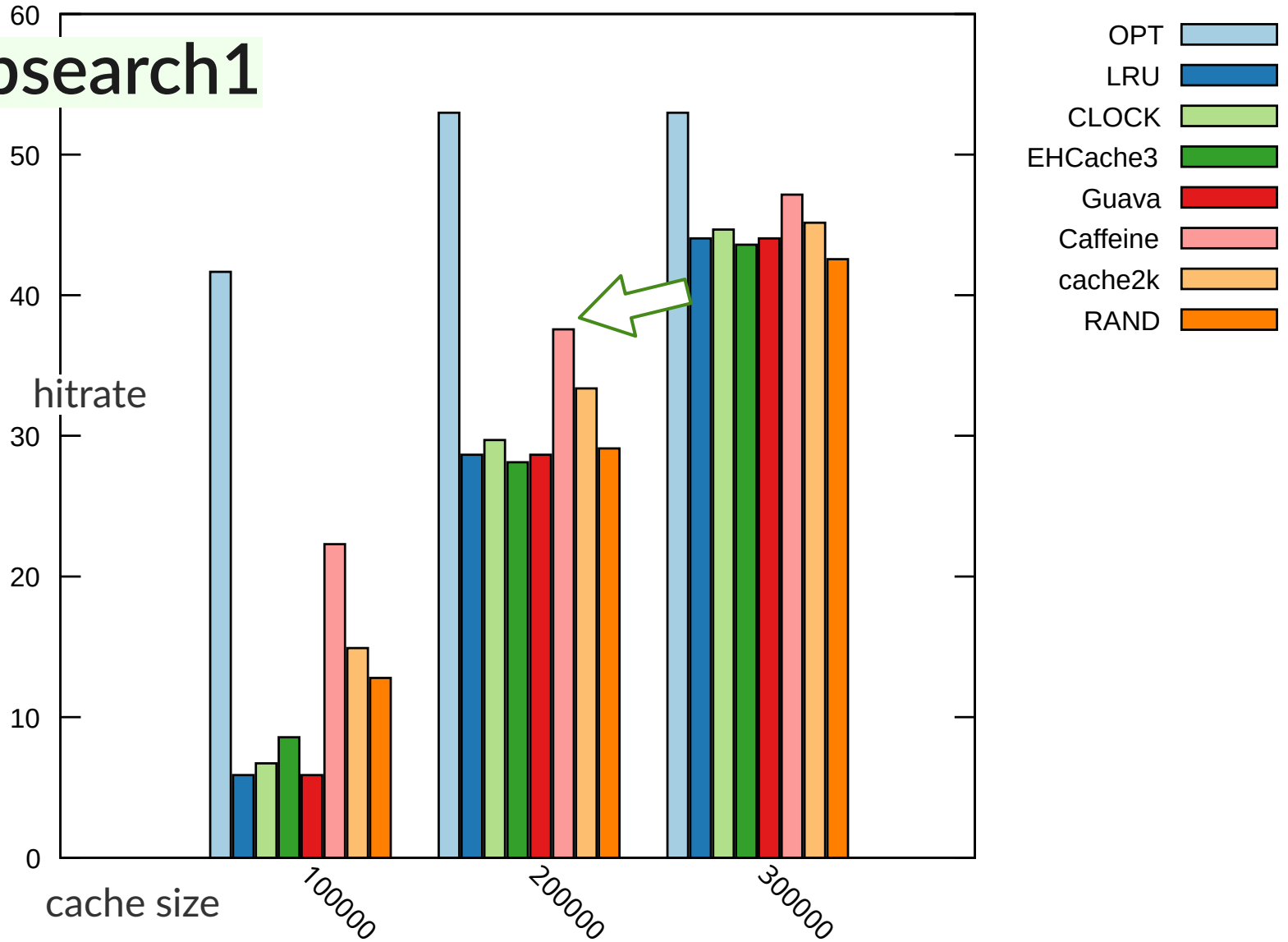
Zipf10k


100
90
80
70
60
50
40
30
20
10
0
hitrate
500
2000
8000
cache size
OPT
LRU
CLOCK
EHCache3
Guava
Caffeine
cache2k
RAND

OrmAccessBusytime



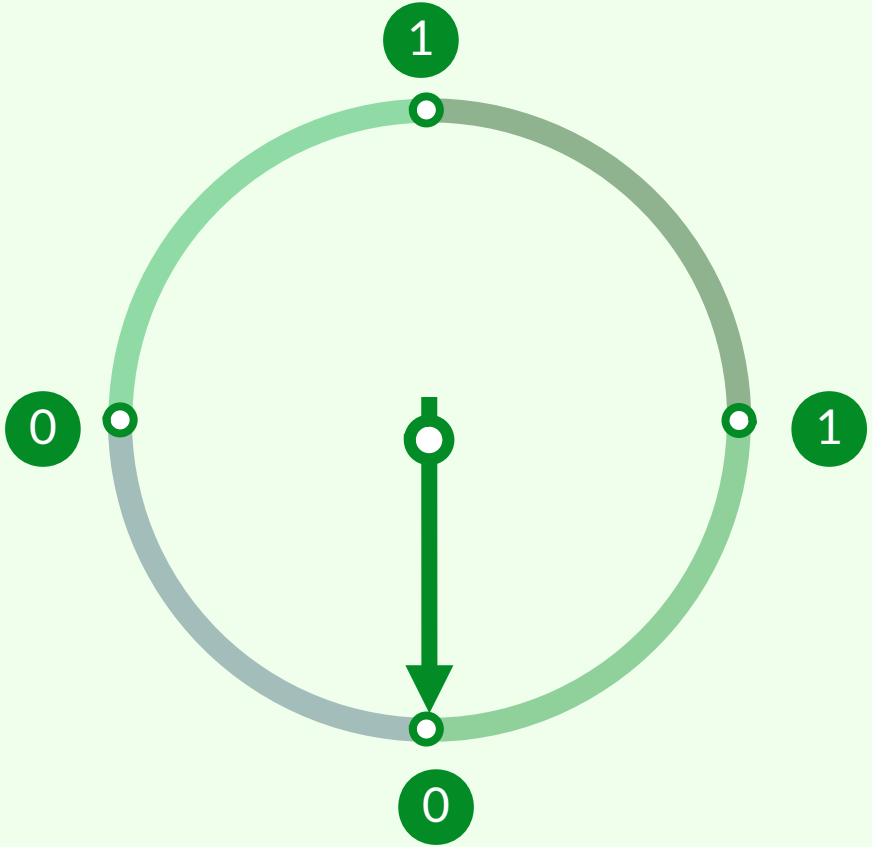
UMassWebsearch1



Results

- Eviction Improvements Caffeine and cache2k
- Varying results depending on access sequences / workloads

But… Isn‘t Clock O(n)?!



Accademic Objection:

Time for eviction grows linear with the cache size

Yes, in theory….

Cache2k and Clock-Pro

Improved algorithm

- Uses counter instead of reference bit
- Heuristics to reduce intensive scanning, if not much is gained

battle tested

- test with a random sequence at 80% hitrate, results in the following average entry scan counts:
  - 100K entries: 6.00398
  - 1M entries: 6.00463
  - 10M entries: 6.00482

=> Little increase, but practically irrelevant.

Technical Overview I

	Guava	Caffeine	EHCache3	cache2k
Latest Version	26	2.6.2	3.6.1	1.2.0.Final
JDK compatibility	8+	8+	8+	6+
sun.misc.Unsafe	-	X	X	-
Hash implementation	own	JVM CHM	old CHM	own
Single object per entry	-	-	-	X
Treeifications of collisions	-	X	-	-
Metrics for hash collisions	-	-	-	X
Key mutation detection	-	-	-	X

Technical Overview II

	Guava	Caffeine	EHCache3	cache2k
Eviction Algorithm	Q + LRU	Q + W-TinyLFU	„Scan8“	Clock-Pro+
Lock Free Cache Hit	Lock free	Lock + Wait free	Lock free	Lock + Wait free
Limit by count	X	X	X	X
Limit by memory size	-	-	X	-
Weigher	X	X	-	X
JCache / JSR107	-	X	X	X
Separete API jar	-	-	-	X

Try cache2k?

- Open Source, Apache 2 Licence
- On Maven Central
- Info and User Guide at: *https://cache2.org*
- JCache support
- Compatible with Android or pure Java
- Runs with hibernate, Spring, datanucleus
- Compatible with future Java version because of no `sun.misc.Unsafe` magic

Summary

- LRU is simple but outdated
- Caffeine and cache2k use modern eviction algorithms and have (mostly) better eviction efficiency than LRU
- Caffeine likes to have cores
- EHCache3 likes to have memory
- cache2k optimizes on a fast/fastest access path for a „cache hit“ while having reasonable eviction efficiency
- Modern hardware needs modern algorithms
- Faster caches allows more fine grained caching

Keep Tuning! Questions?

Jens Wilke
@cruftex
cruftex.net

Up Next

Appendix / Backup Slides

Simple Cache – Part I

```
public class LinkedHashMapCache<K,V>
  extends LinkedHashMap<K,V> {

  private final int cacheSize;

  public LinkedHashMapCache(int cacheSize) {
    super(16, 0.75F, true);
    this.cacheSize = cacheSize;
  }

  protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
    return size() >= cacheSize;
  }
}
```

Simple Cache – Part II (thread-safe)

```
public class SynchronizedLinkedHashMapCache<K,V> {

  final private LinkedHashMapCache<K,V> backingMap;

  public void put(K key, V value) {
    synchronized (backingMap) {
      backingMap.put(key, value);
    }
  }

  public V get(K key) {
    synchronized (backingMap) {
      return backingMap.get(key);
    }
  }

}
```

Simple Cache – Part II – Thread Safety

```
public class SynchronizedLinkedHashMapCache<K,V> {

  final private LinkedHashMapCache<K,V> backingMap;

  public void put(K key, V value) {
    synchronized (backingMap) {
      backingMap.put(key, value);
    }
  }

  public V get(K key) {
    synchronized (backingMap) {
      return backingMap.get(key);
    }
  }

}
```

Simple Cache – Part III – Partitioning/Segmentation

```
public class PartitionedLinkedHashMapCache<K,V> {

  final private int PARTS = 4;
  final private int MASK = 3;
  final private LinkedHashMapCache<K, V>[] backingMaps =
    new LinkedHashMapCache[PARTS];

  public void put(K key, V value) {
    LinkedHashMapCache<K, V> backingMap = backingMaps[key.hashCode() & MASK];
    synchronized (backingMap) {
      backingMap.put(key, value);
    }
  }

  public V get(K key) {
    LinkedHashMapCache<K, V> backingMap = backingMaps[key.hashCode() & MASK];
    synchronized (backingMap) {
      return backingMap.get(key);
    }
  }

}
```

Example 1: Geolocation Lookup