Java In-Process Caching – Performance, Progress and Pitfalls

17th October 2018
Lightweight Java User Group Munich

Jens Wilke
@cruftex
cruftex.net

About Me

- Performance Fan(atic)
- Author of cache2k

https://cache2k.org

- 70+ answered questions on StackOverflow about Caching
- JCache / JSR107 Contributor
- General manager of a boutique software engineering shop in Munich
- @cruftex / cruftex.net

Content

In-Process Cache Fundamentals

- What is an in-process cache doing
- Why you maybe should not write another one
- Performance comparison
- Technical Overview

Next talk or discussion:

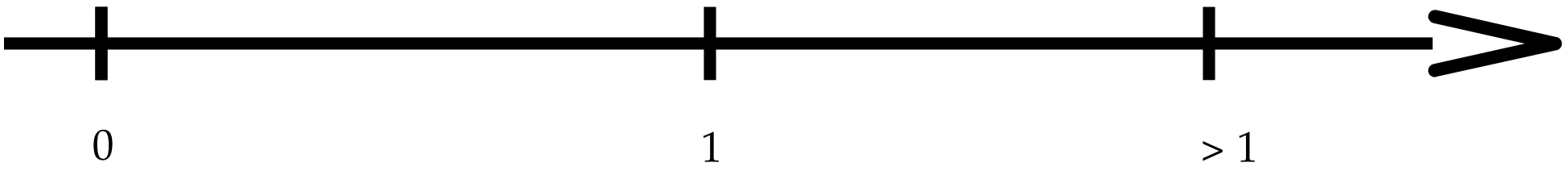
- Features
- APIs

Cache by Wikipedia

In computing, a cache /kæʃ/ kash,[1] is a hardware or software component that stores data so that future requests for that data can be served faster

Web Application

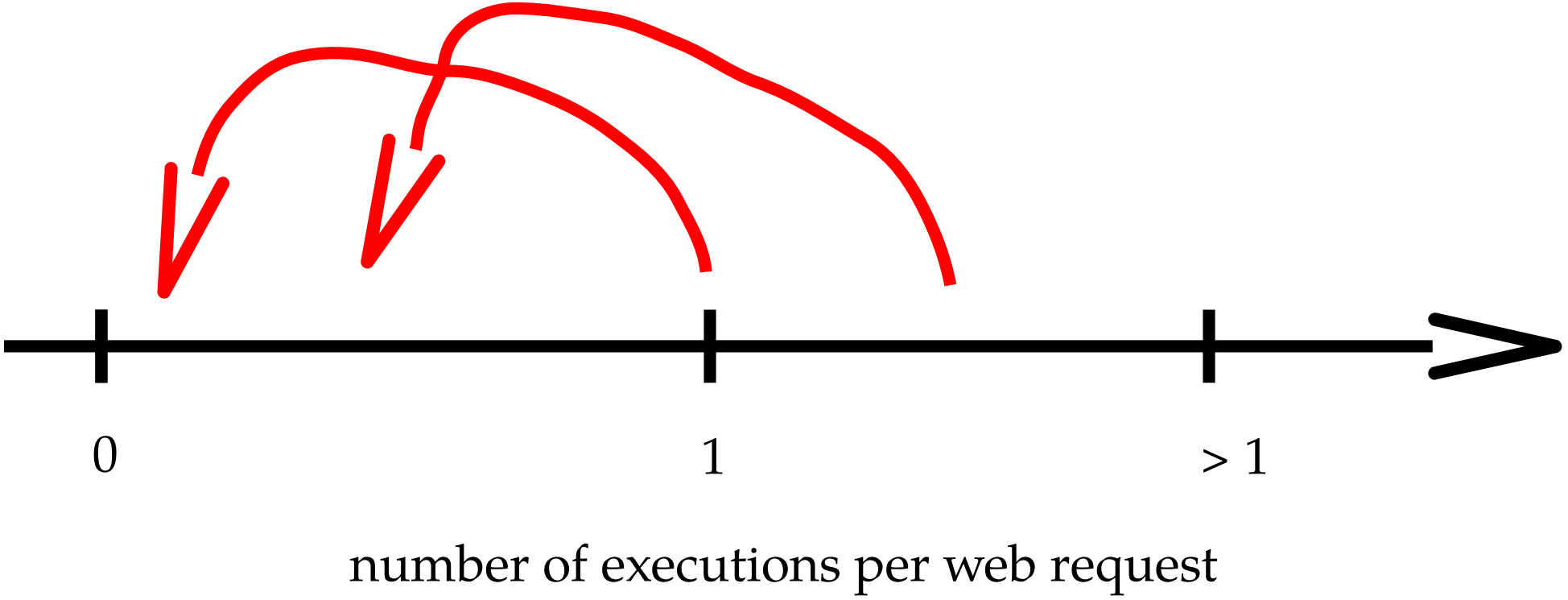
- Every line of code I see I can usually put on the this axis:



number of executions per web request

Web Application

- With a cache we need less executions



Java In-Process Caching

- A tool to tackle the space time tradeoff problem
- Improve UX and lower latency:
  Have hot data as „close to the CPU as possible“
  (on the heap and as object reference)

Whats needed?

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

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

- We need two things for a cache:
  - A hash table
  - An eviction strategy
    - to limit the used memory
    - but keep data that is „hot“

A Simple Cache with LinkedHashMap

```
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;
  }

}
```

Side note….

- We need a cache and all we get from the JDK is the LinkedHashMap?!
- Digging into the JDKs Java code internals we will find a lot of cache implementations for a lot of different things…..

A Simple Cache with LinkedHashMap

What is missing?!

```
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;
  }

}
```

Ooops, Thread Safety!

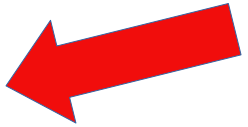
```
public class SynchronizedLinkedHashMapCache<K,V> {

  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);
    }
  }

}
```

A ReadOnly Benchmark (JMH)

```
BenchmarkCache<Integer, Integer> cache;
Integer[] ints;
entryCount = 10_000;

@Setup
public void setup(){
  cache = getFactory().createUnspecialized(entryCount * 10);
  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]);
}
```

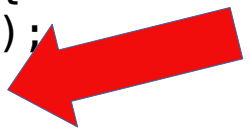
A ReadOnly Benchmark (JMH)

```
BenchmarkCache<Integer, Integer> cache;
Integer[] ints;
entryCount = 10_000;

@Setup
public void setup(){
  cache = getFactory().createUnspecialized(entryCount * 10);
  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]);
}
```



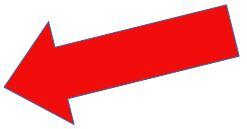
A ReadOnly Benchmark (JMH)

```
BenchmarkCache<Integer, Integer> cache;
Integer[] ints;
entryCount = 10_000;

@Setup
public void setup(){
  cache = getFactory().createUnspecialized(entryCount * 10);
  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]);
}
```



A ReadOnly Benchmark (JMH)

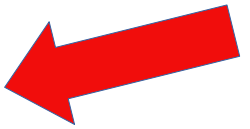
```
BenchmarkCache<Integer, Integer> cache;
Integer[] ints;
entryCount = 10_000;

@Setup
public void setup(){
  cache = getFactory().createUnspecialized(entryCount * 10);
  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]);
}
```



Benchmark Specs

Environment:

- 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



Library Versions:

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

Code is at:

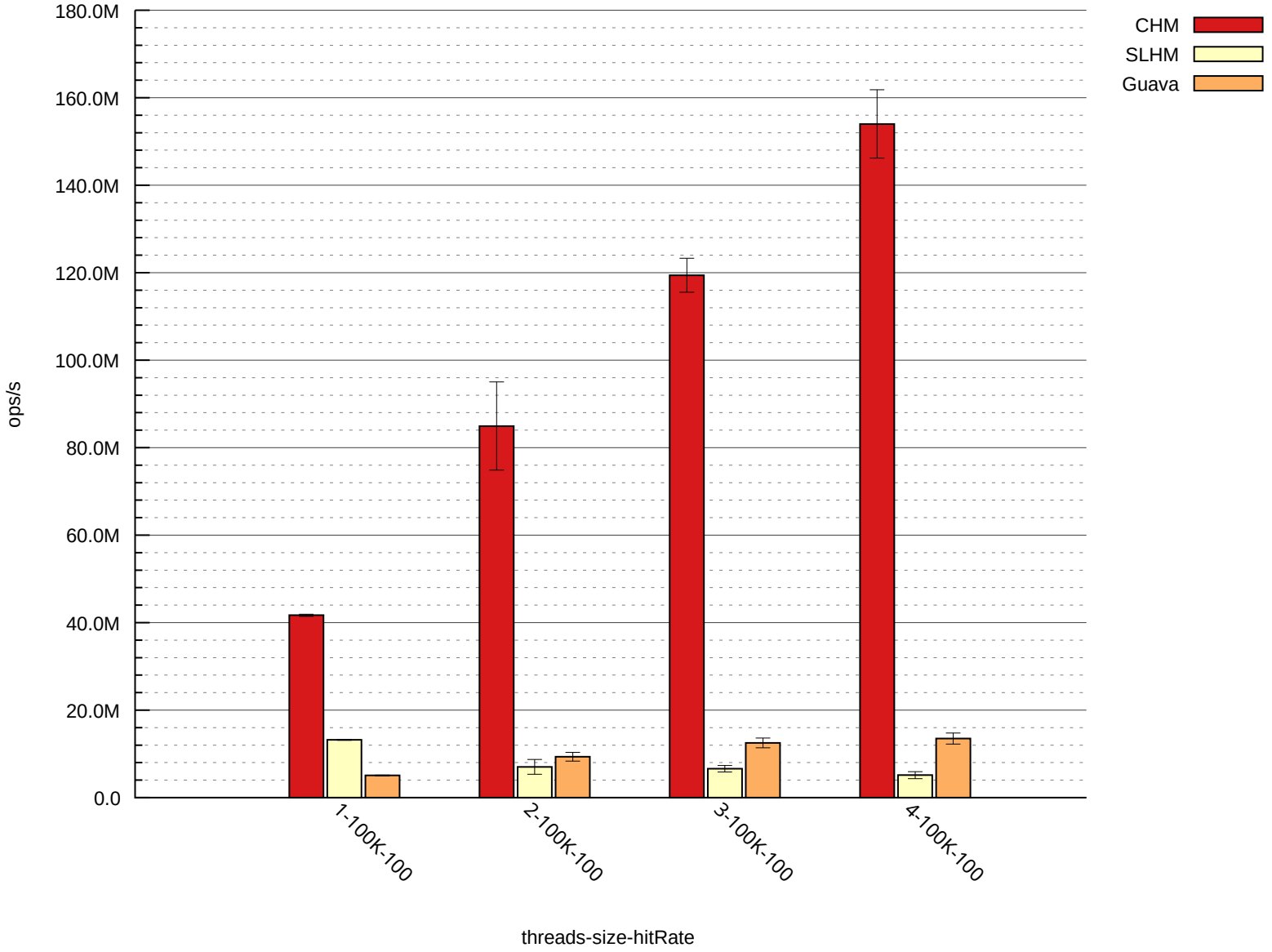
https://github.com/cache2k/cache2k-benchmark

JMH Parameters

- JMH Parameters:
  - 2 forks, 2 warmup iterations, 3 measurement iterations, 15 second iterations times

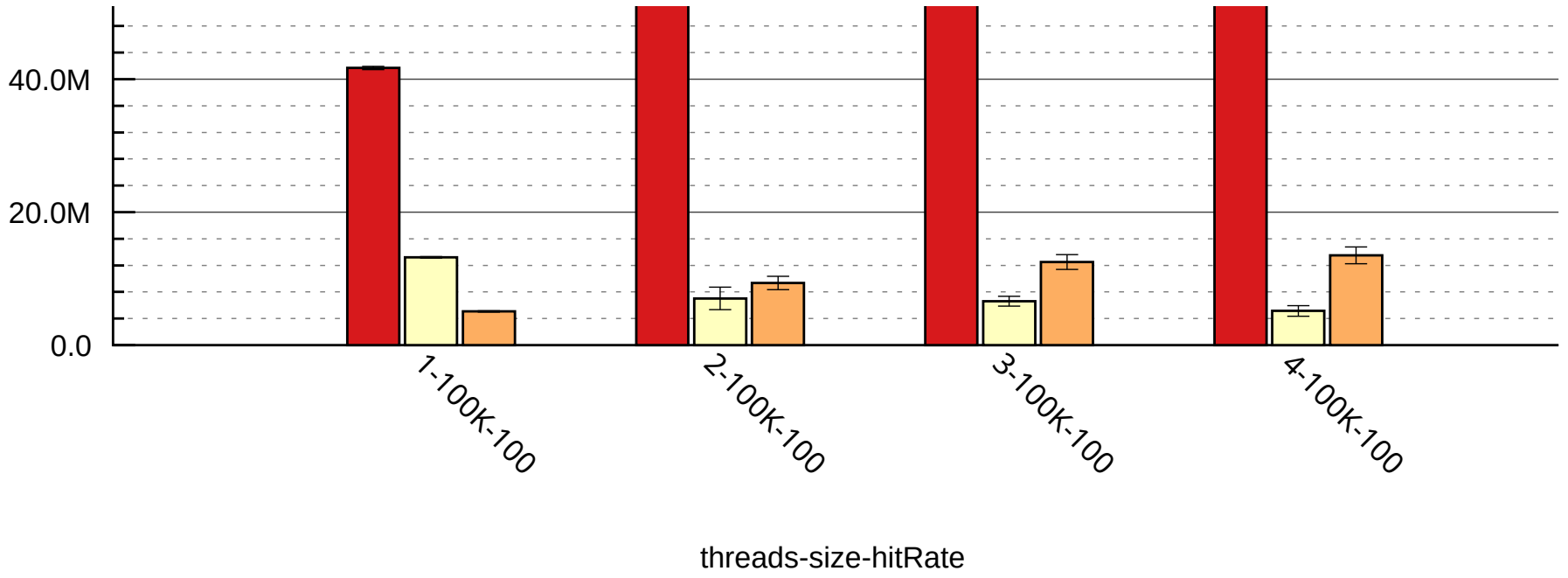
  => 6 measurement iterations
- Graphs show the confidence interval
- Confidence interval is at 99.9% confidence level!

Read Only Benchmark Results



Read Only Benchmark Results

- Red: CuncurrentHashMap
- Yellow: SynchronizedLinkedHashMapCache
- Orange: Google Guava Cache

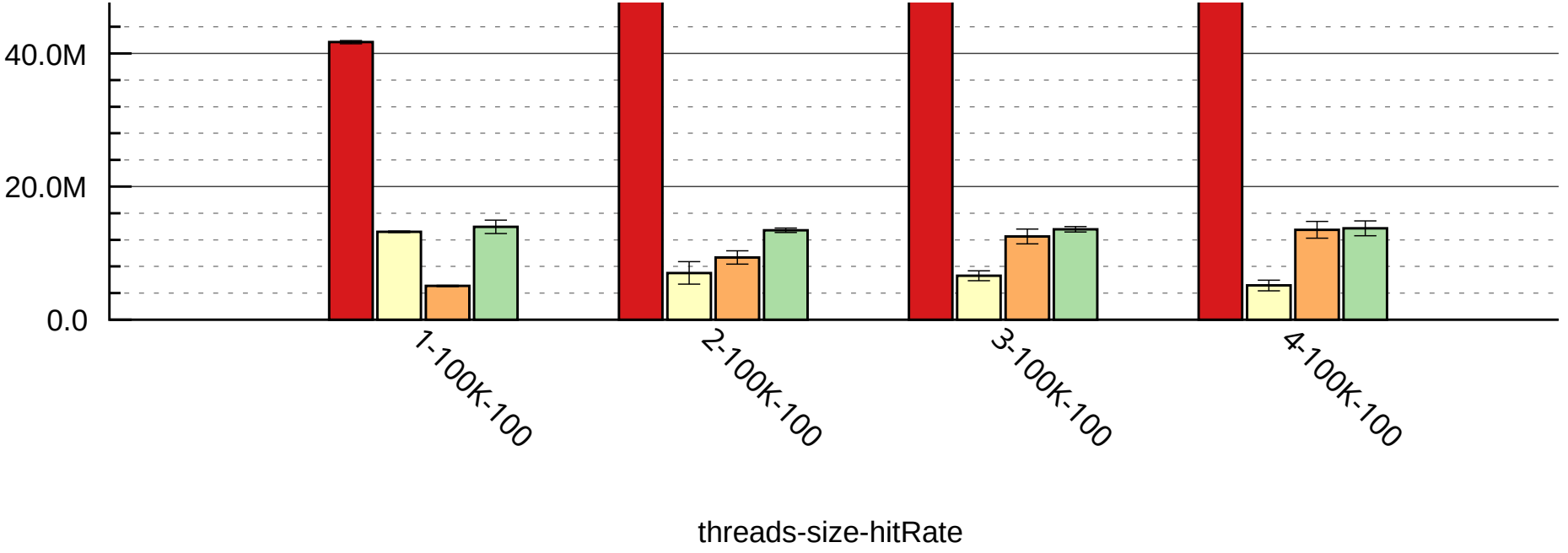


More Concurrency: Partitioning

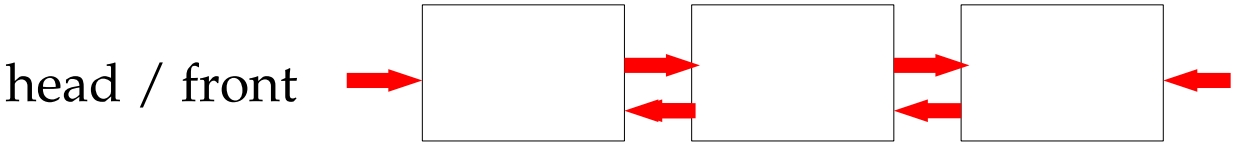
```
public class PartitionedLinkedHashMapCache<K,V> {

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

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

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

}
```

Read Only Benchmark Results II

- Red: CuncurrentHashMap
- Yellow: SynchronizedLinkedHashMapCache
- Orange: Google Guava Cache
- Green: PartionionedLinkedHashMap



LRU – Least Recently Used



1. put(1, x) — 1 | | | — insert new head

2. put(2, x) — 2 | 1 | | — insert new head

3. get(1) — 1 | 2 | | — move to front

4. put(3, x) — 3 | 1 | 2 — insert new head

5. put(4, x) — 4 | 3 | 1 — 2 > Insert new head remove tail

LRU Properties

- 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 are looking for:

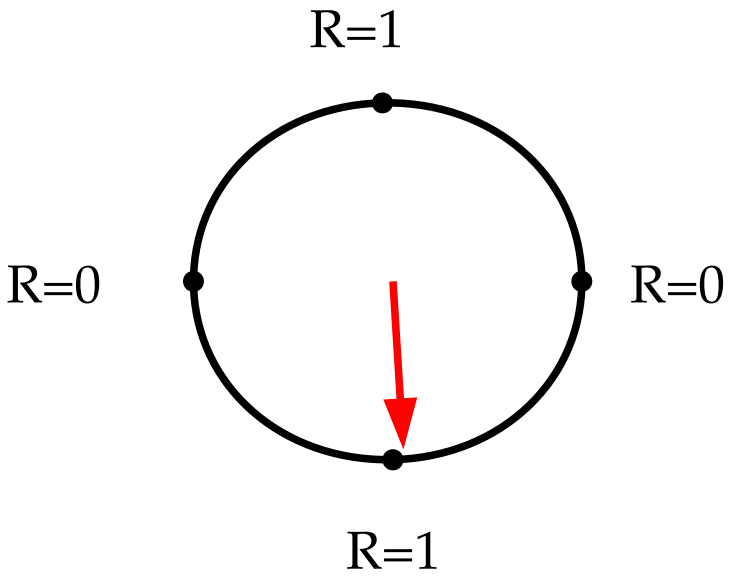
- Reduce CPU cycles for the read operation and do more costly things for eviction later
- Also take frequency into account, keeping more frequently accessed objects longer in the cache

Yes, there are some, see:

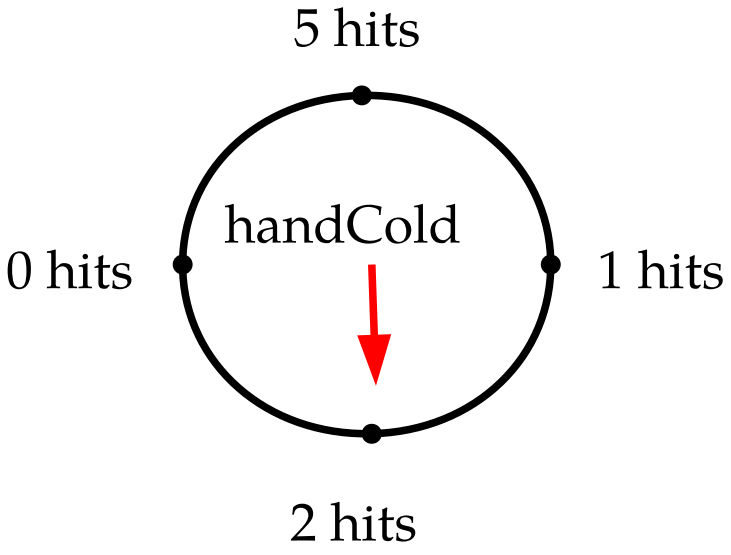
Wikipedia:Page_replacement_algorithm
Lots of scientific papers!

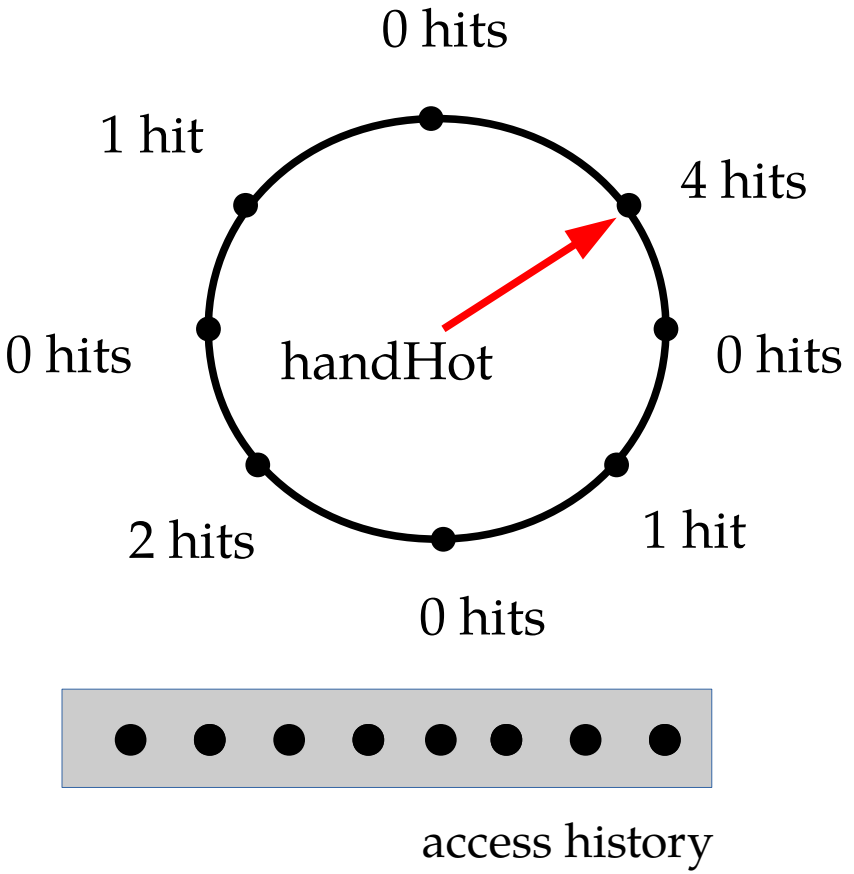
Clock

- Read: Set the reference bit
- Eviction: Scan at clock hand:
  - Referenced? Clear reference and move to the next
  - Not-Referenced? Evict!



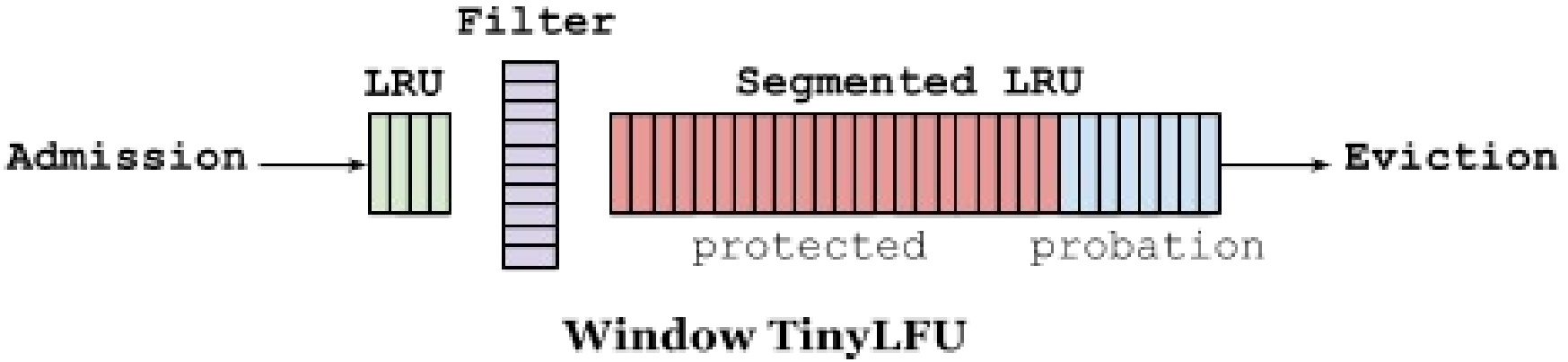
Clock Pro + (cache2k)





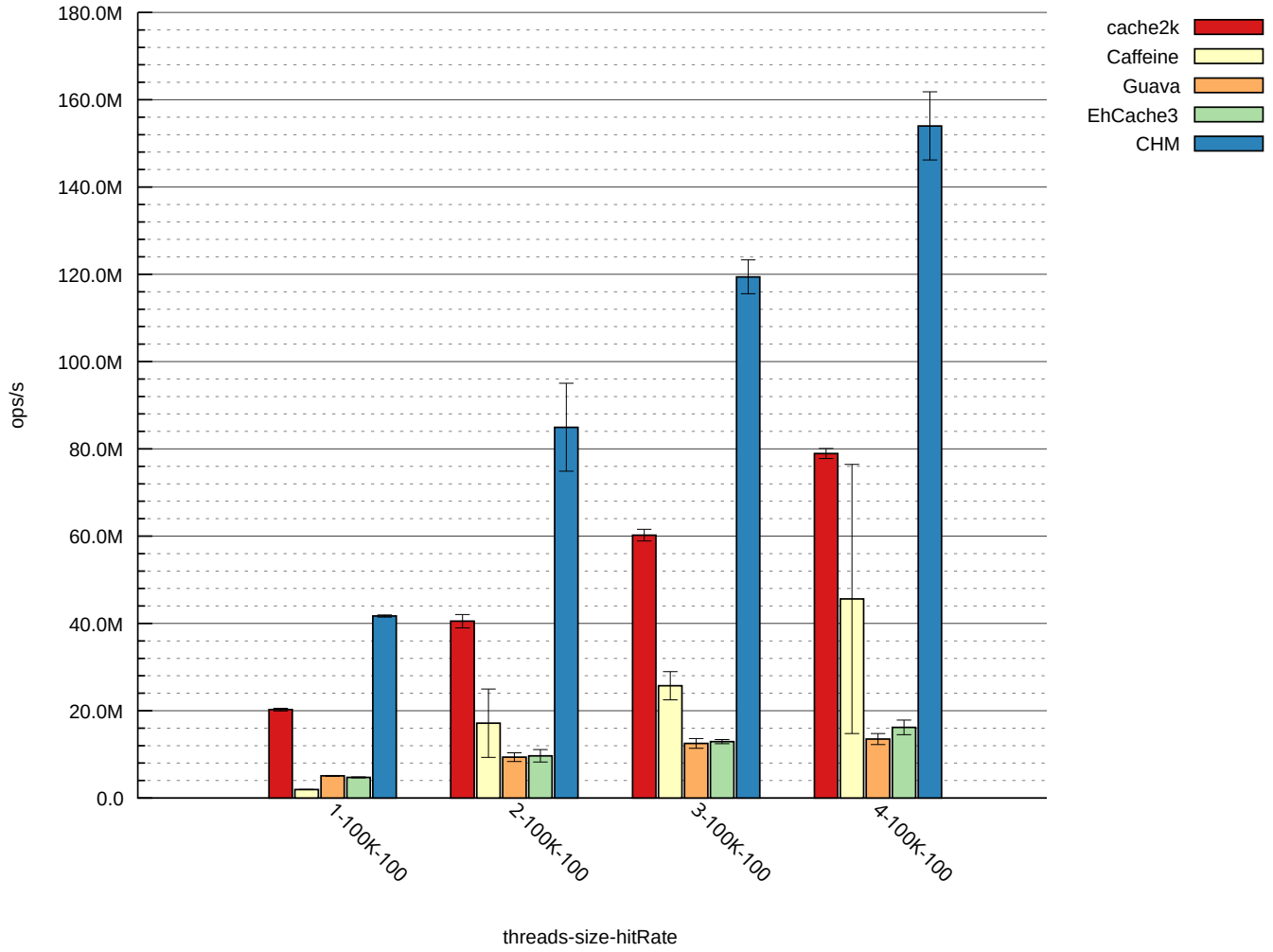
Queue +W-Tiny-LFU (Caffeine)

- Remember cache access in a queue
- Update data structures for eviction alogrithm in a separete thread

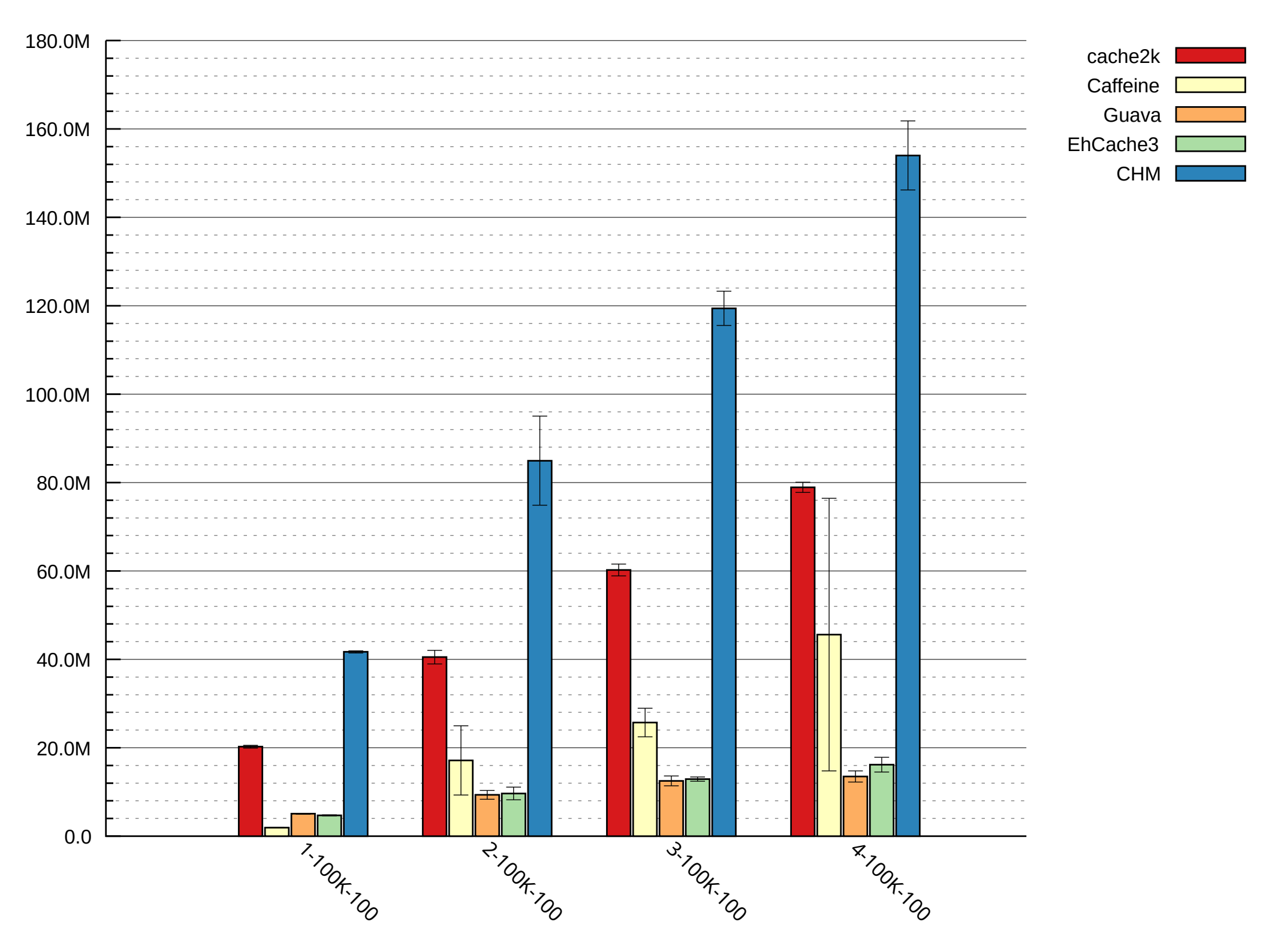


Will It Blend?

Read Only Benchmark Results III



Bigger….


180.0M
160.0M
140.0M
120.0M
100.0M
80.0M
60.0M
40.0M
20.0M
0.0
1-100K-100
2-100K-100
3-100K-100
4-100K-100
cache2k
Caffeine
Guava
EhCache3
CHM

Pretty fast!

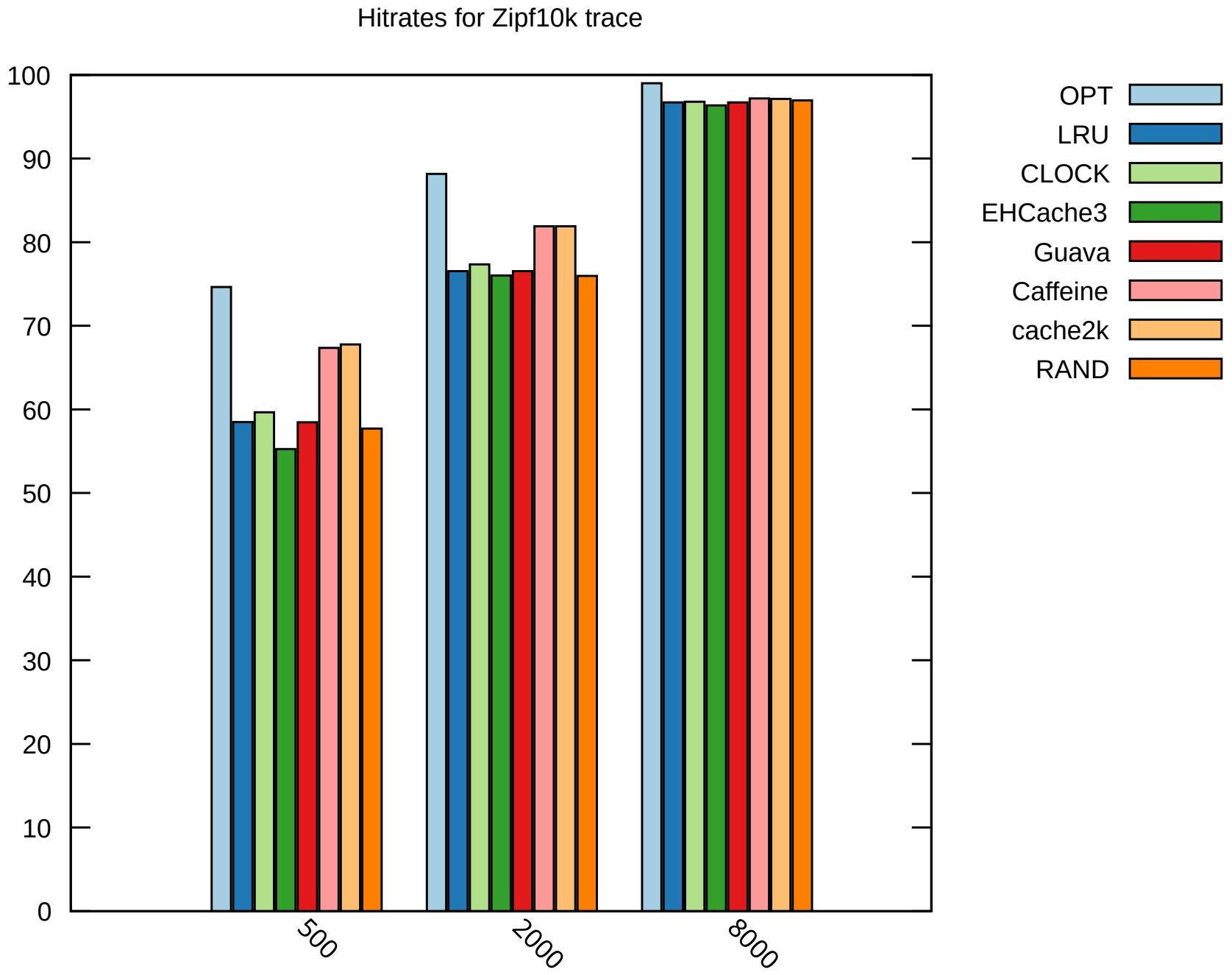
But what about eviction efficiency?

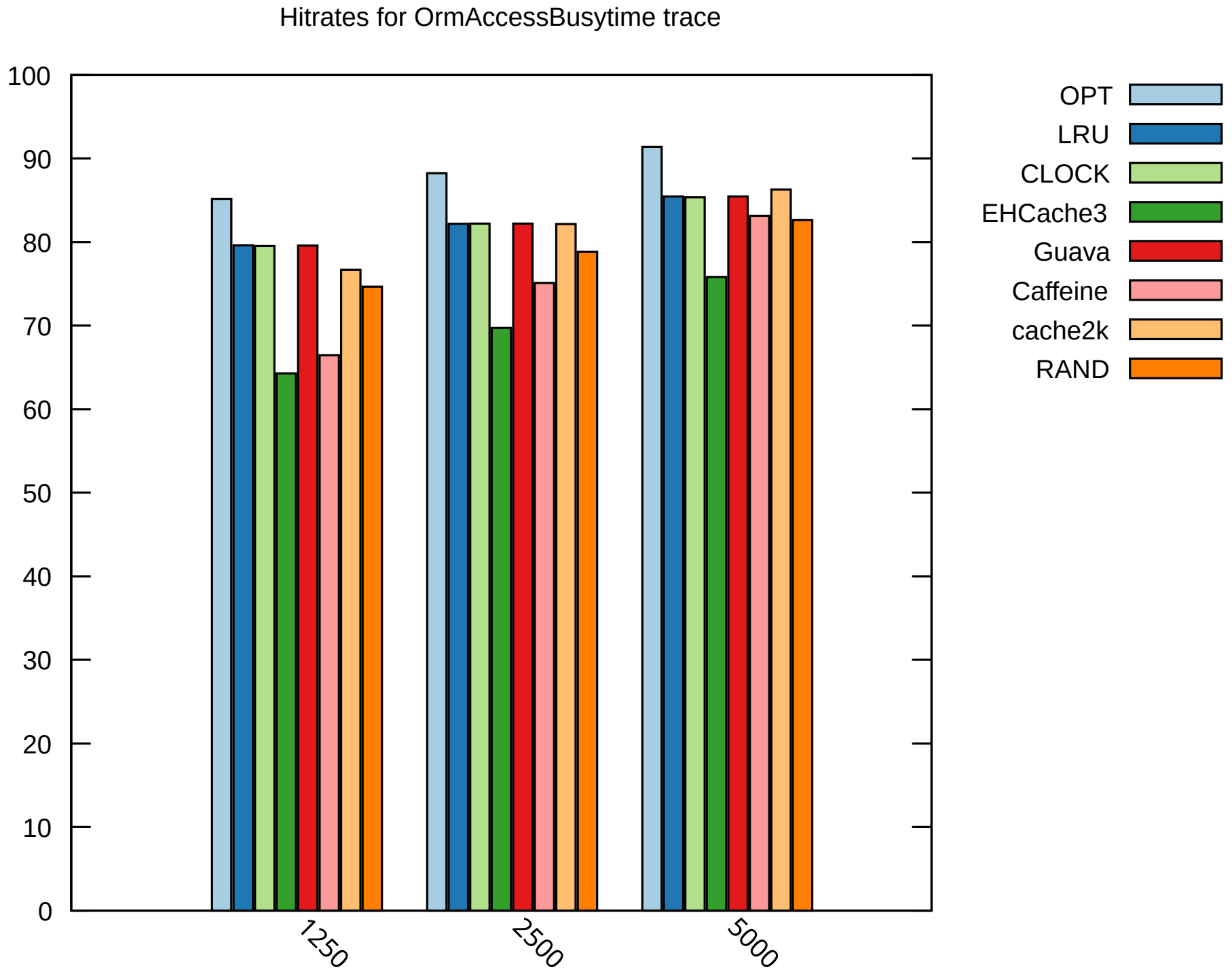
Benchmarking Eviction Quality

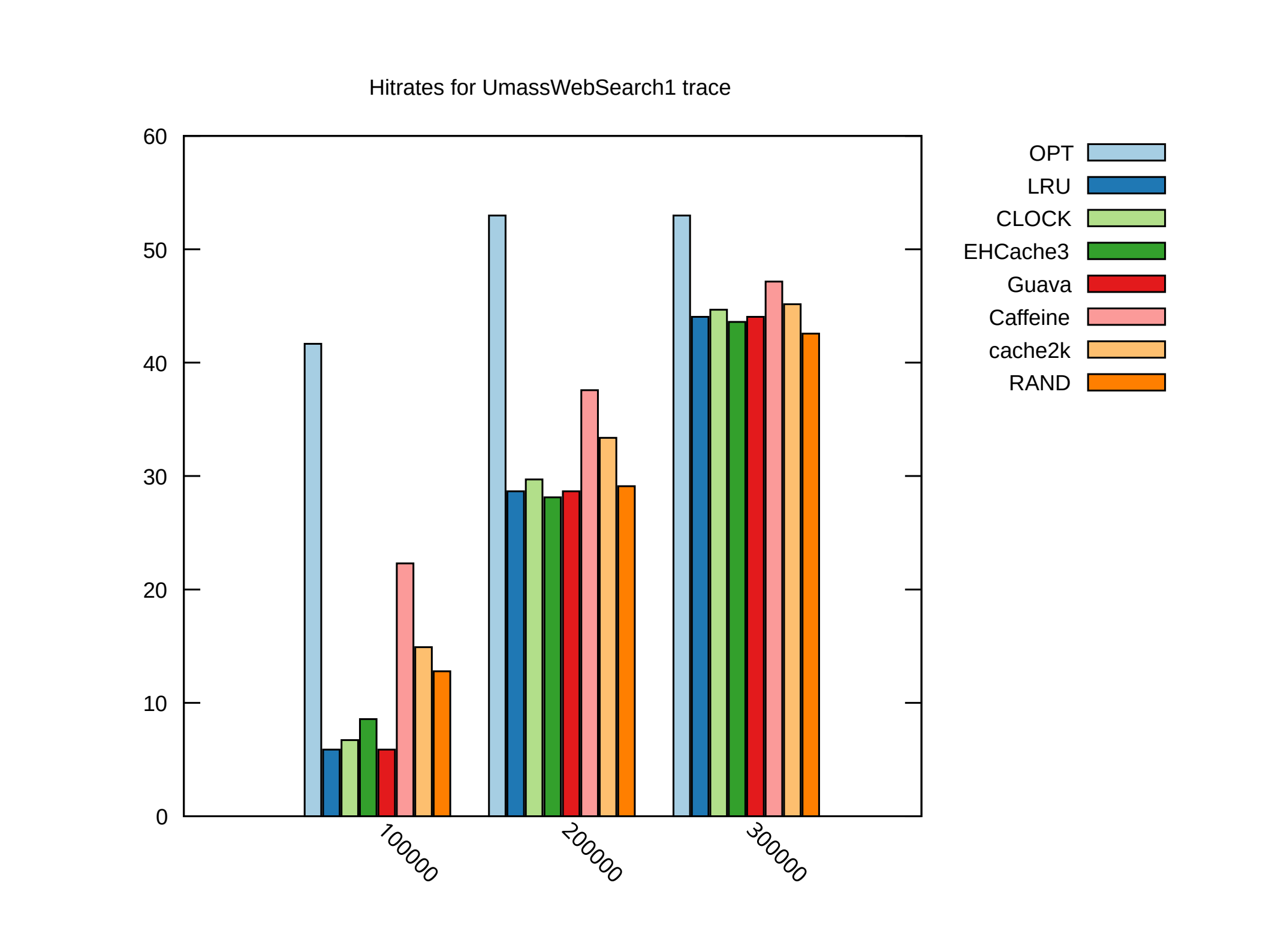
- Collect access sequences (traces)
- Replay the access sequence on a cache and count hits and misses

More information about the traces in the blog article:

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


Hitrates for Zipf10k trace
100
90
80
70
60
50
40
30
20
10
0
500
2000
8000
OPT
LRU
CLOCK
EHCache3
Guava
Caffeine
cache2k
RAND

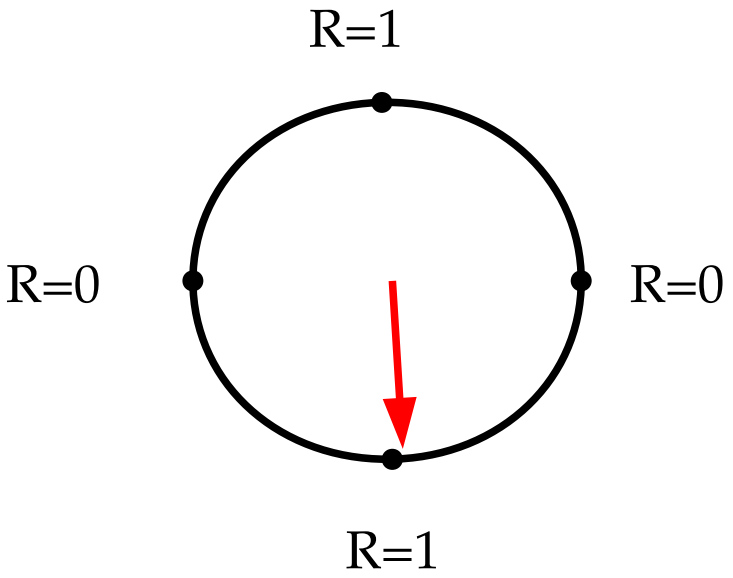

Hitrates for OrmAccessBusytime trace
100
90
80
70
60
50
40
30
20
10
0
1250
2500
5000
OPT
LRU
CLOCK
EHCache3
Guava
Caffeine
cache2k
RAND


Hitrates for UmassWebSearch1 trace
60
50
40
30
20
10
0
100000
200000
300000
OPT
LRU
CLOCK
EHCache3
Guava
Caffeine
cache2k
RAND

Eviction Result

New algorithms from
Caffeine and cache2k look good :)

Great?! No! Accademic Objection: Clock is O(n)!



Cache2k and Clock-Pro +

- Improved and battle tested Clock-Pro algorithm
  - Uses counter instead of reference bit
  - Heuristics to reduce intensive scanning, if not much is gained
- E.g. 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.

Hint / Confession

- The shown performance benchmark only reads and only covers the „hit“ case
- Yes, when eviction is needed, or with read/write the performance benefit of cache2k becomes less

More benchmark results at: https://cruftex.net

All benchmarks are in:

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

Summary

- LRU is simple but there are better options
- Caching != LRU
- Caffeine and cache2k use modern eviction algorithms and have better eviction efficiency than LRU
  Unfortunately not in every studies scenario, but mostly
- 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

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	-	DEV
JCache / JSR107	-	X	X	X
Separete API jar	-	-	-	X

How „usable“ is cache2k?

- Guava, Caffeine, EhCache, cache2k have a comparable set of basic features (e.g. listeners, read through / CacheLoader, expiryAfterWrite / time to live)
- cache2k is tested and integrated with:
  - Hibernate (via JCache)
  - datanucleus (via JCache)
  - Spring Framework
- Extras in cache2k:
  - Sophisticated epxiry + refresh ahead
  - Resilience / exception handling
- Missing in cache2k:
  - Eviction listener, weigher, async cache loader (next version)
  - expireAfterAccess / time to idle (available via JCache, but slow)

Project Links

- Benchmarks:
  https://github.com/cache2k/cache2k-benchmark
- Cache implementations:
  - https://github.com/google/guava/wiki/CachesExplained
  - https://github.com/ben-manes/caffeine
  - http://www.ehcache.org/
  - https://cache2k.org

Questions?

Thanks &
Enjoy Live!