A multicore performance mystery solved

Peter Sestoft (sestoft@itu.dk)

IT University of Copenhagen, Denmark

Version 1.0 of 2017-03-19

Abstract: We investigate how an "obvious optimization" can make a concurrent Java program considerably slower. We demonstrate that the culprit is the hardware: false sharing of CPU cache lines and the attendant cache coherence protocol overhead. This underscores the vast difference between the simple machine models adequate for understanding sequential program performance and the much subtler machine models needed to understand parallel program performance. We show that functional programming with Java streams sidesteps some of this complexity.

We systematically measure the (very considerable) scalability problems caused by false sharing. We demonstrate with a realistic data structure case study that for sufficiently large server hardware classical scalability techniques such as lock striping are ineffective unless one pays attention to cache structure also.

1 The mystery

The January 2017 exam [4, 5] in Practical Concurrent and Parallel Programming (PCPP) at the IT University of Copenhagen studied parallelization of k-means clustering. There was an apparent mystery where seemingly improved code would be considerably slower than the corresponding unimproved code.

The unimproved parallel code, called KMeans2P, looks like this, with some details left out. The "`let taskCount parallel tasks do`" syntax is pseudocode for using Java's executor framework to create and start `taskCount` parallel tasks, each processing a segment `[from...to-1]` of the `points` array:

```
while (!converged) {
  // Assignment step: put each point in exactly one cluster
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      myCluster[pi] = closest(points[pi], clusters);
  }
  // Update step: recompute mean of each cluster
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      myCluster[pi].addToMean(points[pi]);
  }
  converged = true;
  for (Cluster c : clusters)
    converged &= c.computeNewMean();
}
```

In the assignment step, eight tasks collaborate to go through 200,000 points and for each such point `points[pi]` find the cluster (of which there are 81) whose mean is closest to the point, and store a reference to that cluster in `myCluster[pi]`. The call `closest(p, clusters)` returns the cluster whose mean is closest to point `p`; see page 3.

In the update step, the same number of tasks collaborate to traverse the `points[]` and `myCluster[]` arrays at the same time, and add each point `points[pi]` to the (previously computed) closest cluster's `sumx` and `sumy` and `count` fields (see below), taking a lock on the cluster for each field update.

Now it would seem simpler and faster to just do the `addToMean(p)` operation in the assignment step, instead of letting the assignment step write to the `myCluster` array and the update step traverse it later on. Then the update step's only work would be to compute the new means for the 81 clusters, and one can get rid of the `myCluster` array.

The resulting shorter parallel code, called KMeans2Q, would look like this:

```
while (!converged) {
  // Assignment step: put each point in exactly one cluster
  let taskCount parallel tasks do {
    final int from = ..., to = ...;
    for (int pi=from; pi<to; pi++)
      closest(points[pi], clusters).addToMean(points[pi]);
  }
  // Update step: recompute mean of each cluster
  converged = true;
  for (Cluster c : clusters)
    converged &= c.computeNewMean();
}
```

Indeed the simpler KMeans2Q code is 5 percent faster than KMeans2P *when executed sequentially*, with `taskCount` equal to 1. Here's the average wall clock time of 10 runs, each processing 200,000 points into 81 clusters in 108 iterations of the `while` loop:

Class	Time (sec)
KMeans2P	4.240
KMeans2Q	4.019

However, when running multiple parallel tasks with `taskCount` equal to 8, the "optimized" KMeans2Q is 70 percent slower than the apparently wasteful KMeans2P which first writes the array and then reads it:

Class	Time (sec)
KMeans2P	1.310
KMeans2Q	2.234

This is for a 4-core (and hyperthreading, so 8 hardware threads) shared-memory CPU, an Intel i7-4870HQ running MacOS 10.12.3 and Oracle's Java SE Hotspot 1.8.0_101-b13.

On a large Windows 10 server with an Intel Xeon E5-2680 v3 that has 2 x 12 cores (and hyperthreading, so 48 hardware threads) at 2.5 GHz, the slowdown from KMeans2P to KMeans2Q is even more remarkable when `taskCount` is 48:

Class	Time (sec)
KMeans2P	0.853
KMeans2Q	6.587

Thus the "optimized" KMeans2Q program is a whopping 7.7 times slower than the initial KMeans2P.

2 Explaining the slowdown

The reason for the slowdown is somewhat subtle: it seems to be caused by the CPU's *cache coherence protocol* and *false sharing* of cache lines.

The Point class is immutable:

```
class Point {
  public final double x, y;
  ...
}
```

The nested Cluster class has four mutable fields and some methods:

```
static class Cluster extends ClusterBase {
  private volatile Point mean;
  private double sumx, sumy;
  private int count;
  public synchronized void addToMean(Point p) {
    sumx += p.x;
    sumy += p.y;
    count++;
  }
  public synchronized boolean computeNewMean() { ... }
}
```

The `addToMean` method takes a lock for threadsafe update of the `sumx`, `sumy` and `count` fields. The `mean` field is updated only by the `computeNewMean` method, and is volatile for visibility of updates even without locking. (In the KMeans2P and KMeans2Q use contexts shown above this does not really matter because the Java executor framework guarantees that whatever was written by the assignment tasks is visible to the update tasks and vice versa).

The method `closest(p, clusters)` traverses the `clusters` array and returns the cluster whose `mean` field is closest to point `p`:

```
private static Cluster closest(Point p, Cluster[] clusters) {
  Cluster bestCluster = null;
  double bestDist = Double.POSITIVE_INFINITY, dist;
  for (Cluster c : clusters) {
    dist = p.sqrDist(c.mean);
    if (dist < bestDist) {
      bestCluster = c;
      bestDist = dist;
    }
  }
  return bestCluster;
}
```

The KMeans2P assignment step reads the `points` array, the `clusters` array, and the Point objects referred by the clusters' `mean` fields. It writes to the `myCluster` array, each assignment task writing only to its own 25,000 element segment of that array. The assignment step takes no locks at all.

The KMeans2P update step reads the `points` array, the `myCluster` array, each cluster's `mean` field, and the `x` and `y` fields of the referred-to Point objects. It updates the 81 clusters' `sumx`, `sumy` and `count` fields after taking the cluster's lock. Each of the 81 locks is taken on average 200,000/81 = 2469 times.

The KMeans2Q assignment step also reads the `points` array and the `clusters` array and takes the 81 cluster locks the same number of times (and it avoids writing and reading the `myCluster` array, and KMeans2Q traverses the `points` array just once). Hence the slowdown cannot be attributed to additional lock taking, increased lock contention, increased memory usage or the like. In fact, memory usage decreases by 1.6 MB, the size of the `myCluster` array.

The problem in KMeans2Q is that the 200,000 memory updates to the 81 cluster objects' `sumx`, `sumy` and `count` fields are interleaved with the 81 $\times$ 200,000 read accesses to the cluster objects' `mean` fields.

Namely, it seems plausible that the JVM runtime layouts a Cluster object's fields like this:

`mean`	`sumx`	`sumy`	`count`

or perhaps some other permutation of the four fields, but in any case the fields fit into a single cache line, which is 64 bytes on the i7-4870HQ and the Xeon E5-2680. Whenever a CPU core wants to write the `sumx`, `sumy` or `count` field during a call to the `addToMean` method, that core will have to put the cache line into the E (for Exclusive) and then M (for Modified) state of the MESI cache coherence protocol [2, section 2]. This will cause the entire cache line to be invalidated in all other cores' caches, invalidating also their cached copy of the `mean` field — just because they are on the same cache line. Thus the cache line containing `mean` repeatedly switches between the E, M and I (for Invalid) states. This is unavoidable, and happens also in the KMeans2P update step, but there it is quite harmless because the `mean` fields are not accessed at all. During the KMeans2Q assignment step, however, the `mean` field of each cluster will be read 200,000 times (once for each point). The cache line invalidations caused by `addToMean` give rise to cache misses on subsequent read accesses which will therefore be much slower, by a factor of maybe 100, than an L1 cache hit.

To validate the hypothesis that false cache line sharing causes the slowdown, we can pad the Cluster object with dummy fields in the hope that they prevent the `mean` field from being on the same cache line as the `sumx`, `sumy` and `count` fields at runtime:

```
static class Cluster extends ClusterBase {
  private volatile Point mean;
  private long dummyy1, dummyy2, dummyy3, ..., dummyy16;
  private double sumx, sumy;
  private int count;
  private long dummy1, dummy2, dummy3, ..., dummy16;
  ...
}
```

The expectation is that the padding produces a memory layout like this, which ensures that the `mean` field is on a different cache line that `sumx`, `sumy` and `dummy`:

`mean`	dy1	...	dy16	`sumx`	`sumy`	`count`	d1	...	d16

And indeed the padding dramatically improves the KMeans2Q performance on both the i7 and the Xeon, actually making their performance on the i7 indistinguishable:

Class	i7 (sec)	Xeon (sec)
KMeans2P	1.344	1.055
KMeans2Q padded	1.342	1.469

So indeed it seems that false cache line sharing is involved in the slowdown. Note that the dummy fields work only because the JVM runtime does not optimize them away, which it safely could, and because it apparently does not reorder the fields aggressively. In Java, there is an experimental field annotation @sun.misc.Contended that is supposed to request such cache line padding in a more robust way, but here its effect was rather modest and achieved nowhere near the same speedup as the manual padding.

The reason for having two groups of dummy fields is that we want to prevent one cluster's `mean` field from sharing a cache line with the previous cluster's `count` field, which could happen if the cluster objects are allocated next to each other in the heap:

mean	sumx	sumy	count		mean	sumx	sumy	count

3 Functional stream programming sidesteps the problem

There seems to be a simple way to avoid these subtleties altogether: Use Java functional parallel streams instead of side-effects, tasks and locks. Here is a k-means implementation, called KMeans3P, using Java 8 streams and functional programming:

```
while (!converged) {
  { // Assignment step: put each point in exactly one cluster
    final Cluster[] clustersLocal = clusters; // For capture in lambda
    Map<Cluster, List<Point>> groups =
      Arrays.stream(points).parallel()
            .collect(Collectors.groupingBy(p -> closest(p, clustersLocal)));
    clusters = groups.entrySet().stream().parallel()
      .map(kv -> new Cluster(kv.getKey().getMean(), kv.getValue()))
      .toArray(Cluster[]::new);
  }
  { // Update step: recompute mean of each cluster
    Cluster[] newClusters =
      Arrays.stream(clusters).parallel().map(Cluster::computeMean)
            .toArray(Cluster[]::new);
    converged = Arrays.equals(clusters, newClusters);
    clusters = newClusters;
  }
}
```

with completely immutable clusters and no side effects in methods:

```
static class Cluster extends ClusterBase {
  private final List<Point> points;
  private final Point mean;
  public Cluster(Point mean, List<Point> points) {
    this.mean = mean;
    this.points = Collections.unmodifiableList(points);
  }
  public Cluster computeMean() {
    double sumx = points.stream().mapToDouble(p -> p.x).sum(),
           sumy = points.stream().mapToDouble(p -> p.y).sum();
    Point newMean = new Point(sumx/points.size(), sumy/points.size());
    return new Cluster(newMean, points);
  }
}
```

Interestingly, the functional parallel stream version KMeansP3 performs extremely well both on the 4-core i7 and the 24-core Xeon, compared to KMeans2P and the unpadded KMeans2Q:

Class	i7 (sec)	Xeon (sec)
KMeans2P	1.310	0.853
KMeans2Q	2.234	6.587
KMeans3P	1.264	0.553

There are at least three reasons for the excellent parallel performance of this functional stream-based solution. First, the Java 8 stream library and its parallelization are very well engineered and probably play well together with the JVM just-in-time compiler. Second, the absence of side effects in the user code obviously means that it does not cause cache line invalidation, since all shared data are immutable and so the cache lines can remain in the S (for Shared) state of the MESI cache protocol, without any invalidation messages. Third, the JVM runtime system's memory allocator and garbage collector obviously must allocate and initialize objects, and reclaim the memory when the objects die, which requires writing to memory. But apparently allocation and collection have been engineered to do that without incurring much cache coherence overhead, while still permitting a high degree of parallelism.

4 Cache coherence overhead

4.1 More than cache read misses

We hinted in Section 2 that the slowdown from KMeans2P to KMeans2Q might be caused by cache read misses. As we shall see, that is not sufficient to explain all of the slowdown. On the i7, the slowdown is roughly 0.9 second, or 900,000,000 ns. Since there are 200,000 x 81 = 16.2 million `mean` field read accesses, this is a slowdown of roughly 55 ns per read access. Other measurements of single-core RAM read access latency on that machine produce numbers around 80–95 ns per access, so this seems somewhat plausible. But note that the 16.2 million accesses are performed by four different physical cores in parallel, so the actual delay experienced by a core (or thread) trying to read the `mean` field is closer to 220 ns, which is much more than the time required for a read access to RAM, the slowest part of internal memory.

Appendix A shows how to measure memory read access times for all cache levels and RAM, and gives numbers for the i7 and Xeon machines used in this note.

On the Xeon server machine the KMeans2Q slowdown is so large (5.7 seconds or 5.7 billion ns or at least 350 ns per access) that it cannot be ascribed to cache read misses alone; the RAM access time is only 95 ns according to Appendix A.

Thus other overheads from the cache coherence protocol must contribute to the KMeans2Q slowdown. We investigate this in the next section.

4.2 Measuring cache coherence overhead

To measure more systematically the impact of false sharing and hence quantify the overhead caused by cache coherence protocols, consider $n = 1, 2, 4, \ldots, 64$ threads concurrently reading and writing a single common location or multiple thread-specific locations:

1. incrementing an `int` field, an dense `int` array, a padded `int` array;
2. incrementing an AtomicInteger, a dense AtomicIntegerArray, a padded AtomicIntegerArray using `getAndIncrement`;
3. incrementing an AtomicInteger, a dense AtomicIntegerArray, a padded AtomicIntegerArray using `compareAndSet`;
4. taking and releasing a single lock, a sense lock array, a padded long array.

In each case 1–4 we consider three subcases: (a) all n threads read and write the same `int` or AtomicInteger, or compete for acquiring and then releasing the same lock; (b) each of the n threads reads and writes its own `int` or AtomicInteger, or acquires and then releases its own lock object but possibly with false sharing because these resources are dense in memory; and (c) same as (b) but now there should be no false sharing because the resources are scattered in memory, using array padding.

Here is the source code for case (4) (a), multiple concurrent threads calling the `run` method and hence competing to take and release a single shared lock:

```
static class OneLock implements Incrementer {
  final Object lock = new Object();
  public void run(int threadNo) {
    synchronized (lock) { }
  }
}
```

Case (4) (b), multiple concurrent threads calling the `run` method and hence taking and releasing each their own lock, so no lock contention at all, only possibly cache line sharing:

```
static class LockArray implements Incrementer {
  final Object[] locks = new Object[maxThreads];
  {
    for (int i=0; i<locks.length; i++)
      locks[i] = new Object();
  }
  public void run(int threadNo) {
    synchronized (locks[threadNo]) { }
  }
}
```

Case (4) (c), multiple concurrent threads calling the `run` method and hence taking and releasing each their own lock, where the `locks` array and the lock object allocation are padded, so no lock contention and presumably no cache line sharing:

```
static class PaddedLockArray implements Incrementer {
  final int padding = 16;
  final Object[] locks = new Object[maxThreads * padding];
  {
    for (int i=0; i<locks.length; i++)
      locks[i] = new Object();
  }
  public void run(int threadNo) {
    synchronized (locks[threadNo * padding]) { }
  }
}
```

Hence the difference in performance between (b) and (c) illustrates the cache coherence overhead caused by false sharing and only that.

On the Intel Xeon 2x12x2 core server machine, this cache coherence overhead causes a factor 12, 14, 9 and 21 slowdown in the cases (1), (2), (3) and (4) listed above. Case (4) is especially striking, as it shows that false sharing (of lock objects) can far exceed the actual time to acquire and release an uncontended lock; see Figure 1. In particular, for the 16-thread case, the total time for the threads to acquire and release a single highly contended lock 32 million (2^{25}) times is 2.313 seconds, the time for the threads to acquire and release 16 completely uncontended but densely allocated locks is 0.888

seconds, and the time for 16 completely uncontended and padded locks is 0.072 seconds. Thus plain lock striping achieves a speed-up factor of 2.6, and padding to avoid a false sharing achieves a further speed-up factor 12.3, for a total speed-up factor of 31.9. Hence lock striping may be strikingly ineffective unless combined with defenses against false sharing, and especially so if only a small amount of work is protected by the lock taking: if the amount of work per lock taking is large, the extra overhead of false sharing does not matter as much.

The absolute amount of time spent taking the locks is also interesting. With 16 threads, each thread acquires and releases a lock 2 million (or 2^{21}) times, and we can optimistically assume that the 16 threads run concurrently on 16 of the 24 available cores. In that case, when competing for a single shared lock, each thread spends 1103 ns acquiring and then releasing the lock (including time spent transitioning from the Running to the Locking thread stages and back). When the 16 threads have each their own lock, there should be little or no contention, and the time per lock acquisition and release decreases to 423 ns. When these thread-local locks are scattered in memory to avoid false sharing, the time is just 35 ns. This indicates that the cache coherence overhead for lock acquisition and release may be 390 ns on this machine. Other experiments indicate cache coherence overheads of 730 to 770 ns per operation, for AtomicInteger increments and CAS operations, respectively.

The code used in this section is in file TestCpuCacheTimes.java, and data and graphs in cpucachetimes.xls.

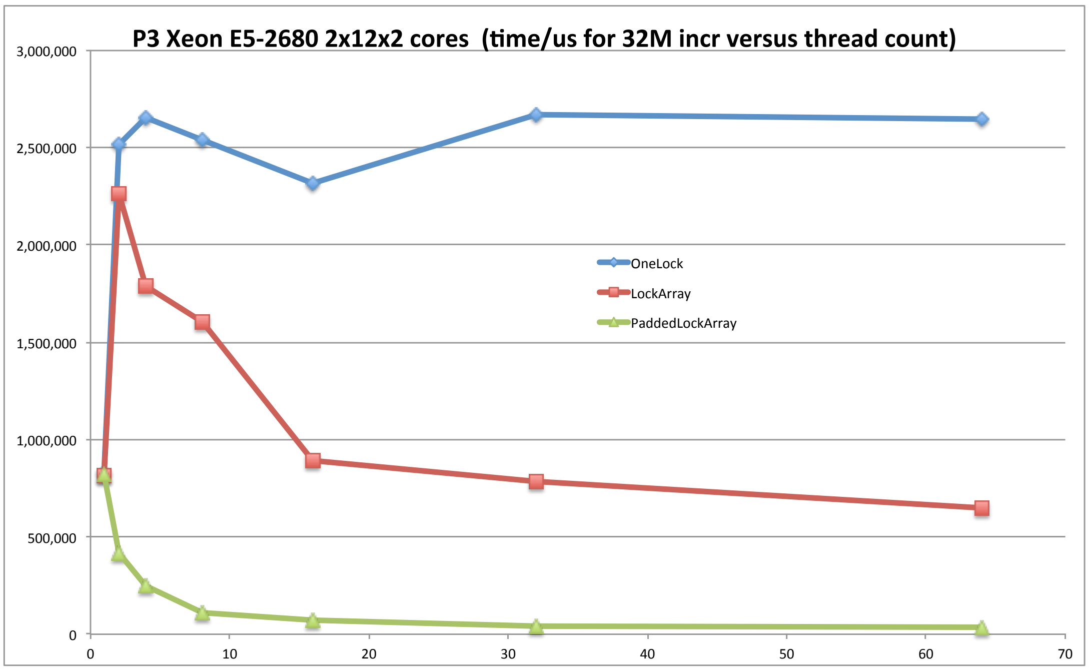

Figure 1: Wall clock execution time in microseconds for 32 million lock operations shared among n threads, as a function of n. For a single shared lock (blue curve) the time is independent of n when $n > 1$; hence no scalability at all. For a lock array (red curve) there is one lock per thread and therefore no lock contention, only cache line interference. For the padded lock array (green curve) there is neither lock contention nor cache line interference.

5 Improving a concurrent hashmap by avoiding false sharing

We can improve the PCPP course StripedWriteMap using these insights. The course presents a case study of concurrent hash maps, including one that uses lock striping and wait-free read access. That implementation uses a dense array of lock objects and a dense AtomicIntegerArray for the sizes of each stripe. An operation such as `put` that modifies a stripe will first take that stripe's lock:

```
class StripedWriteMap<K,V> implements OurMap<K,V> {
  private final Object[] locks;
  private final AtomicIntegerArray sizes;
  ...
  public StripedWriteMap(int bucketCount, int lockCount) {
    this.locks = new Object[lockCount];
    this.sizes = new AtomicIntegerArray(lockCount);
    for (int stripe=0; stripe<lockCount; stripe++)
      this.locks[stripe] = new Object();
    ...
  }
  public V put(K k, V v) {
    final int h = getHash(k), stripe = h % lockCount;
    synchronized (locks[stripe]) {
      ...
  }
  ...
}
```

Such lock striping improves scalability because multiple threads can manipulate distinct stripes concurrently. But the dense allocation of the locks (and stripe size components) means that false cache line sharing could be detrimental to scalability.

Based on this insight, we make a new version of the implementation, padding those two arrays (thus also scattering the lock objects in the heap) by inserting 15 unused locks and 15 unused size components between any two actually used locks and sizes.

```
class StripedWriteMapPadded<K,V> implements OurMap<K,V> {
  private final Object[] locks;
  private final AtomicIntegerArray sizes;
  private final static int padding = 16;
  ...
  public StripedWriteMapPadded(int bucketCount, int lockCount) {
    this.locks = new Object[lockCount * padding];
    this.sizes = new AtomicIntegerArray(lockCount * padding);
    for (int stripe=0; stripe<lockCount * padding; stripe++)
      this.locks[stripe] = new Object();
    ...
  }
  public V put(K k, V v) {
    final int h = getHash(k), stripe = h % lockCount;
    ...
    synchronized (locks[stripe * padding]) {
      ...
    }
  }
  ...
}
```

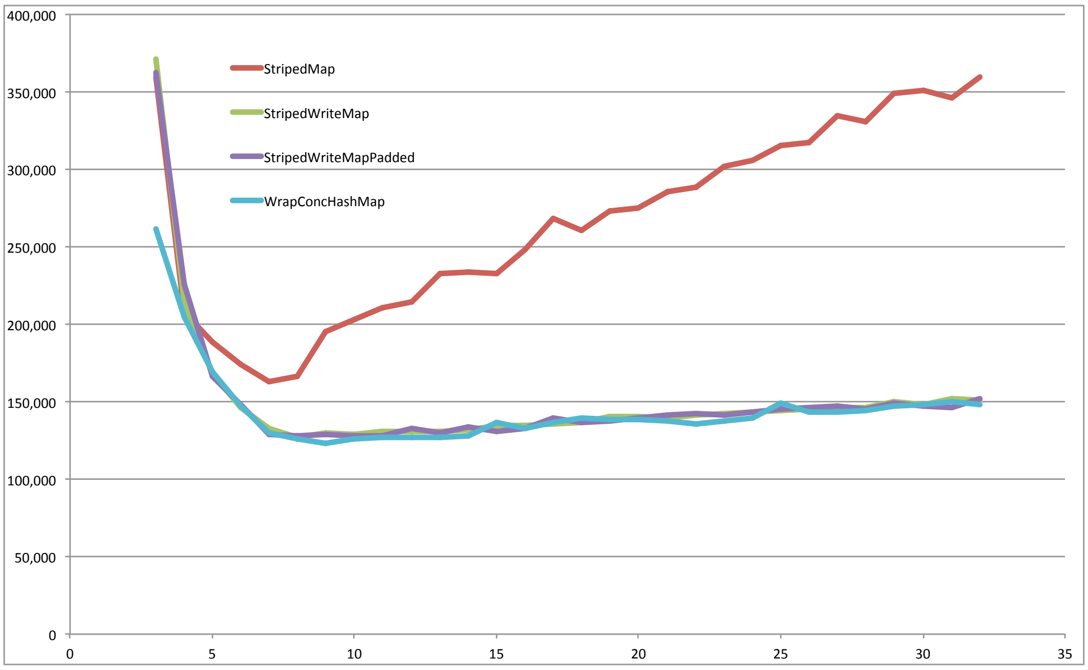

Figure 2: Wall-clock execution time in microseconds for 1.6 million hashmap operations shared between n threads, as a function of n, on Intel i7-4870HQ with 4x2 cores. Using lock striping (red curve) performs poorly beyond $n = 8$ threads. Lock striping with wait-free reads (green), lock striping with padding of the lock array and wait-free reads (purple), and the Java 8 library's ConcurrentHashMap all exhibit roughly the same performance. The lock striping implementations used 32 stripes.

On a 4-core (x 2 hyperthreading) Intel i7 laptop, this has no impact on performance; see Figure 2.

However, on a 2x12 core (x 2 hyperthreading) Intel Xeon server, this trivial change speeds up a test scenario by a factor of 1.1 to 1.5 for larger thread counts, and produces a slowdown of only a few percent for low thread counts (perhaps the latter are random fluctuations); see Figure 3.

Thus lock striping itself may be ineffective unless one pays attention to the allocation of the lock objects. Note that the `locks` array itself is not modified after initialization, so what matters is that the lock *objects* are scattered in memory by the padding, and that the references from the `locks` array to those extra objects keep the garbage collector from removing them and compacting those few lock objects actually used. By contrast, the elements of the AtomicIntegerArray `sizes` do get updated, and it is the padding of the AtomicIntegerArray itself that matters.

6 Conclusion

We have seen that a code change that is an "obvious improvement" in a sequential setting can considerably slow down the code when executed in parallel on a multicore architecture.

We identified the cache coherence protocol and especially false cache line sharing as the culprits, and demonstrated that removing the false sharing restores performance.

We measured the cache coherence overhead of multiple concurrent reading and writing threads, and showed that the slowdown far exceeds the slowdown caused by the cache read misses (measured in the appendix).

Finally, inspired by the earlier insights, we improved the scalability of a lock-striping concurrent hashmap implementation simply by avoiding false sharing in the allocation of locks and size fragments.

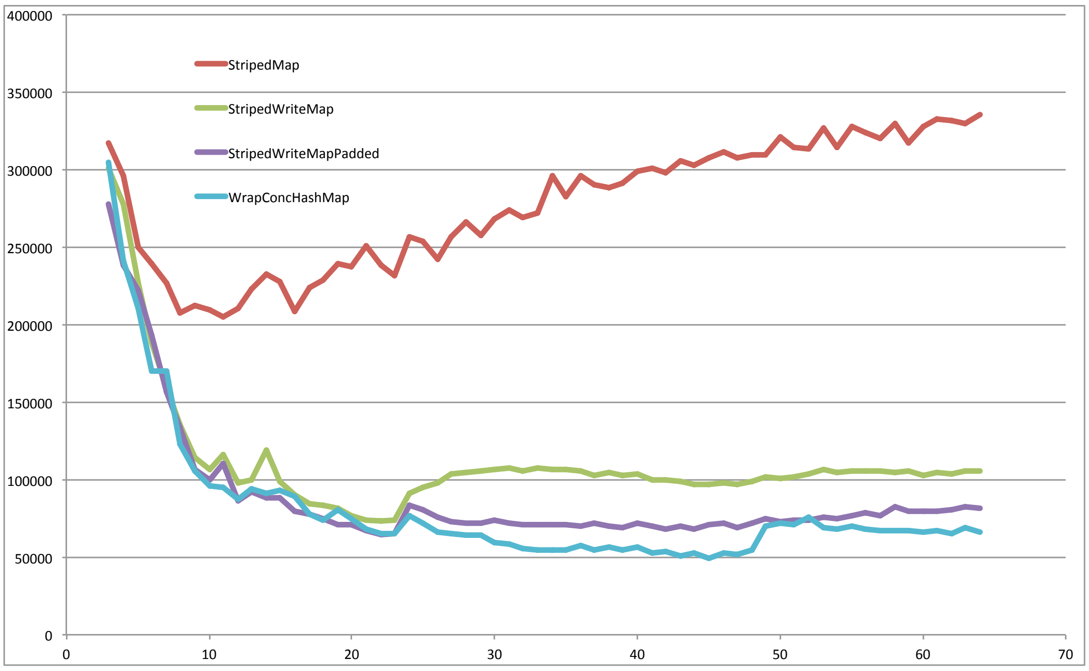

Figure 3: Wall-clock execution time in microseconds for 1.6 million hashmap operations shared between n threads, as a function of n, on Intel Xeon E5-2680 with 2x12x2 cores. Using lock striping (red curve) performs poorly beyond $n = 12$ threads. Lock striping with wait-free reads (green) does not perform well beyond $n = 24$ threads. Lock striping with padding of the lock array and wait-free reads (purple) performs considerably better for such high thread counts, sometimes just as well as the (considerably more complex) Java 8 library ConcurrentHashMap. The lock striping implementations used 32 stripes.

A Measuring read access times

A modern laptop or server CPU has a memory architecture with multiple levels of caches [1, section 3] as illustrated by Figure 4, which also gives some indicative numbers for size and speed of the memory components.

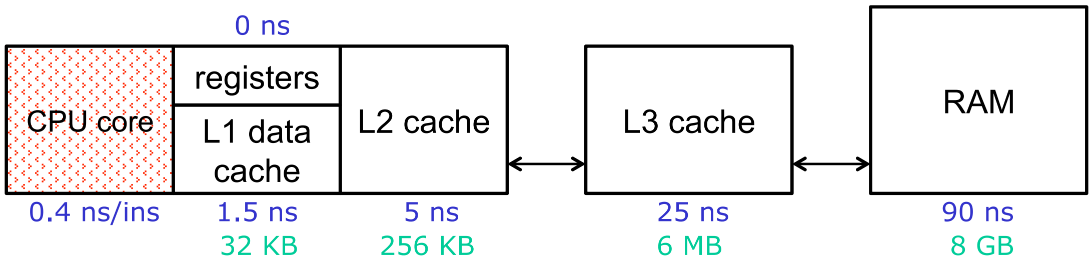

Figure 4: The memory components (caches) of an Intel i7-4870HQ CPU. The L1 and L2 caches are local to a core; the L3 cache and RAM are shared between cores.

In this appendix we present a simple approach to measuring memory read access times, assuming that arrays are stored as contiguous sequences of bytes in memory. For garbage collection reasons this is not necessarily true in Java, but our experimental result fit well with available documentation, and the approach can also be used in C which has a simple and predictable array layout.

A.1 Core measurement loop

The core piece of code to measure memory read times is this method, containing a simple loop:

```
private static double jump(int[] arr) {
  int k = 0;
  for (int j=0; j<33_554_432; j++)
    k = arr[k];
  return k;
}
```

In the setups we consider, the loop always performs 33 million (2^{25}) iterations and array accesses. However, its execution time varies by a factor of more than 60 depending on the contents of the array `arr`. Namely, if the array contains a short cycle of indexes such as 6, 0, 3, 5, 7, 1, 4, 2 then only the first 8 elements of the array will be accessed. Since the 8 elements require only 32 bytes on memory, they will always be in L1 cache, so the loop will effectively measure the L1 cache read access time (plus some overhead for manipulating `j` and array bounds checking). Figure 5 illustrates this situation.

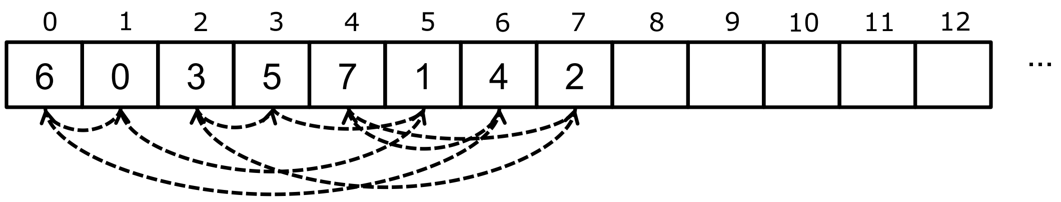

Figure 5: Integer array containing a cyclic index chain of length 8, covering 32 bytes of memory.

If on the other hand the array contains a cyclic chain of indexes that range over 1 million array elements (4 MB memory) then the loop will effectively measure the L3 cache read access time. (That is, provided the indexes do not appear consecutively 1, 2, 3, . . . on the same cache lines; see section A.2).

This table shows the measured time in nanoseconds per iteration of the above `jump` method's loop, as a function of the amount of memory touched:

Memory touched (KiB)	i7-4870HQ	Xeon E5-2680 v3
8	1.4	1.7
16	1.4	1.5
32	1.4	1.5
64	2.5	2.6
128	2.9	3.2
256	3.3	3.5
512	8.0	11.2
1,024	10.2	14.2
2,048	11.4	15.6
4,096	12.9	16.4
8,192	39.6	19.8
16,384	66.0	21.0
32,768	79.2	41.4
65,536	86.0	80.5
131,072	94.5	94.7

The table shows that despite the `jump` loop always performing exactly the same sequence of instructions, the time varies from 1.4 to 94.5 ns depending on the contents of `arr`, that is, by a factor of 67. This is due solely to different memory latencies. The timings are wall-clock measurements obtained using our microbenchmarks methods [3].

The i7-4870HQ has 32 KiB L1 data cache and 256 KiB L2 cache per core, 6,144 KiB L3 cache shared between cores, and 16 GiB RAM. The Xeon E5-2680 v3 has 32 KiB L1 data cache and 256 KiB L2 cache per core, 30,720 KiB L3 cache shared between cores, and 32 GiB RAM. Indeed the table shows that the access time is constant for 8–32 KiB (L1 cache) and reasonably constant for 64–256 KiB (L2 cache) for both machines. Also it jumps considerably between 4096 and 8192 KiB (outgrowing the L3 cache) for the i7, and similarly between 16384 and 32768 KiB for the Xeon.

A.2 How to populate the array of indexes

The array `arr` of used in method `jump` is filled by method `fillArray(arr, S)` which ensures that `arr[0..S-1]` contains a random cyclic chain of indexes of length `S`, starting and ending at `arr[0]`:

```
private static void fillArray(int[] arr, int S) {
  ArrayList<Integer> indexes = new ArrayList<>();
  for (int k=0; k<S; k++)
    indexes.add(k);
  Collections.shuffle(indexes);
  // Make indexes[0] == 0.
  int shift = S - indexes.get(0);
  for (int k=0; k<S; k++)
    indexes.set(k, (indexes.get(k) + shift) % S);
  // Make arr[0] = indexes[1], arr[arr[0]] = indexes[2],
  // ..., arr^k[0] = indexes[k], ..., arr^S[0] = 0;
  // the cyclic chain starting at arr[0] will have length S.
  int i = 0;
  for (int k=0; k<S; k++)
    i = arr[i] = indexes.get((k+1) % S);
}
```

The `fillArray` code is a little intricate, so here is an explanation of the design. The idea is to force the `jump` loop to read ever larger ranges of memory, thus observing the difference between those ranges that fit in L1, L2 and L3 caches, or only in RAM.

Some possibilities and their drawbacks or advantages are:

1. Sequentially reading the elements of an array would not work because reading one element on a cache line (typically 64 bytes) would fetch also the other (typically 15) integer elements on that cache line.

2. Instead one could use a constant stride ≥ 1 as in

   ```
   final int S = arr.length;
   for (int k=0; k<S; k++)
     arr[k] = (k + stride) % S;
   ```

   This would cover the entire array provided `arr.length` and `stride` are mutually prime, but it is too friendly towards the L1 hardware prefetch mechanism, which will recognize the constant stride and prefetch the next item if the `stride` is small enough.

3. Instead one could initialize the array with random indexes like this:

   ```
   for (int k=0; k<S; k++)
     arr[k] = rnd.nextInt(S);
   ```

   and then follow the chain of indexes `for (...)  k = arr[k]`, but one would likely get short chains thanks to the "birthday paradox" and so visit only a small part of the array, maybe held in cache.

4. Instead one could initialize the array with elements `0,1,...,S-1` and randomly shuffle these by `Collection.shuffle()`, but one could still get very short chains, eg there is a non-zero probability that `arr[0]` equals `0`. One could mitigate that risk by reshuffling the arrays until the reference chain is at least 90 percent of the array length, but this is inefficient.

5. An even better approach is to obtain a random permutation of an array `indexes` containing `0,1,...,S-1`, normalize it so that `indexes[0]` equals `0` (by subtracting `indexes[0]` from each element, modulo `S`), and fill the array `arr` with next-indexes from a sequential scan of `indexes`. This will give a cyclic index chain of length `S`.

The last approach is the one implemented by method `fillArray`. The code used in this appendix is in file TestMemoryLatency.java.

References

[1] Ulrich Drepper. What every programmer should know about memory. Technical report, Redhat, 2007. At http://www.akkadia.org/drepper/cpumemory.pdf.

[2] Paul E. McKenney. Memory barriers, a hardware view for software hackers. Technical report, Linux Technology Center, IBM Beaverton, 2010. At http://irl.cs.ucla.edu/~yingdi/web/paperreading/whymb.2010.06.07c.pdf.

[3] Peter Sestoft. Microbenchmarks in Java and C#. Lecture notes, September 2015. At http://www.itu.dk/people/sestoft/papers/benchmarking.pdf.

[4] Peter Sestoft. Examination, Practical Concurrent and Parallel Programming, 10-11 January 2017, January 2017. At http://www.itu.dk/people/sestoft/itu/PCPP/E2016/pcpp-20170110.pdf.

[5] Peter Sestoft. Supporting code for examination in Practical Concurrent and Parallel Programming, January 2017. At http://www.itu.dk/people/sestoft/itu/PCPP/E2016/pcpp-20170110-code.zip.