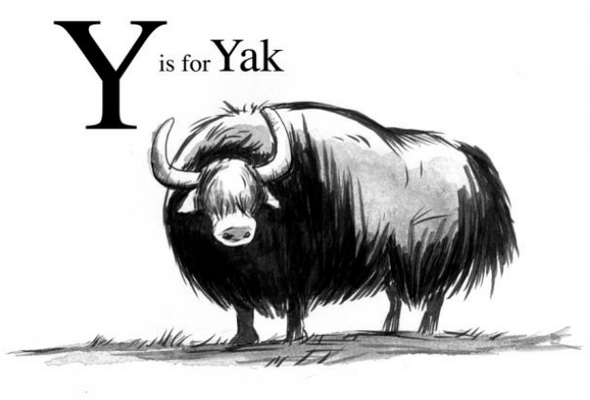
Who am I?

- Nitsan Wakart
- Java Performance Guy @ Azul Systems
- Blogger: http://psy-lob-saw.blogspot.com
- Twitter: @nitsanw

Code/Slides for this talk: https://github.com/nitsanw/QueueEvolution

Queues Evolution



Circular Array

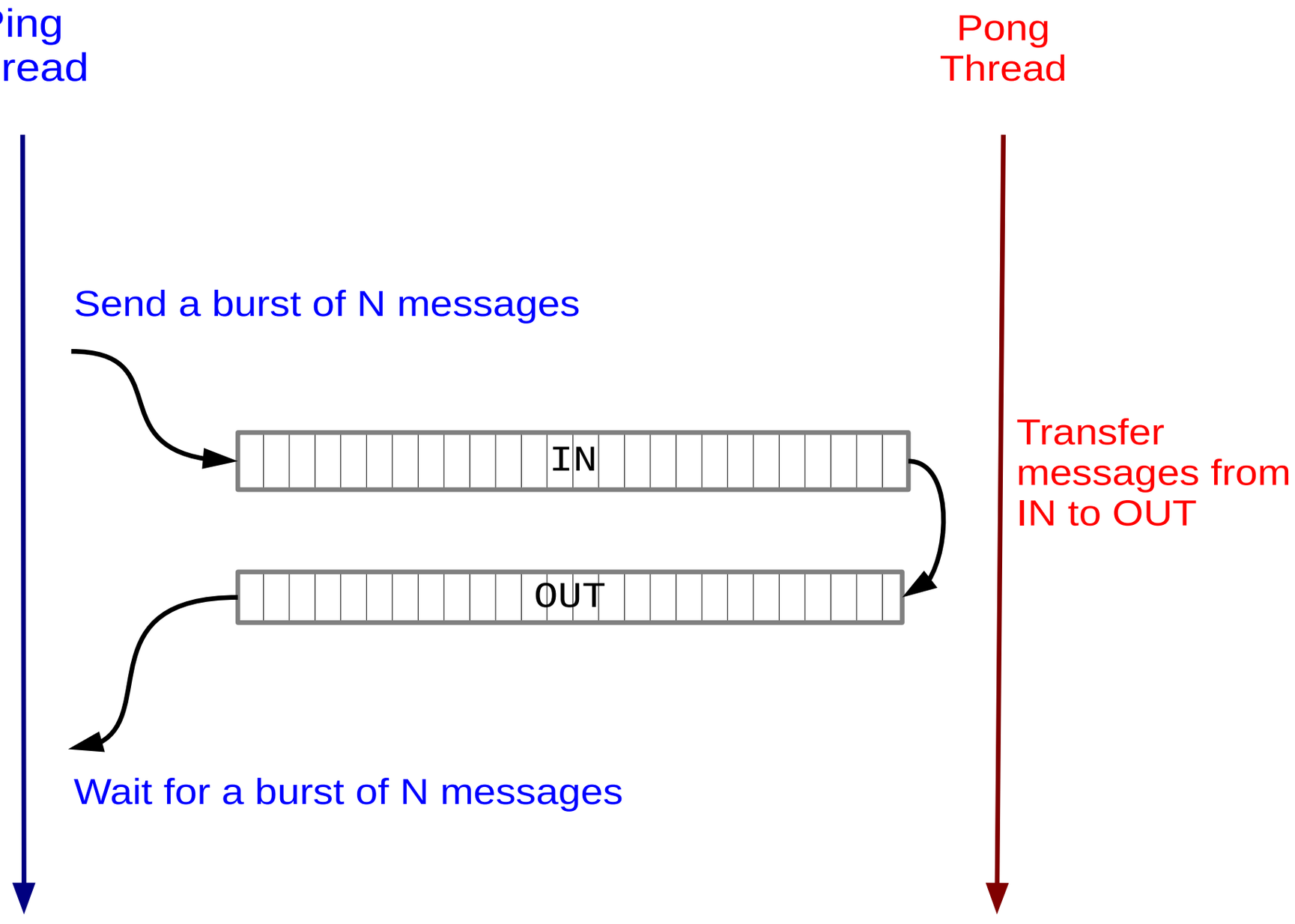
Lamport

Thompson

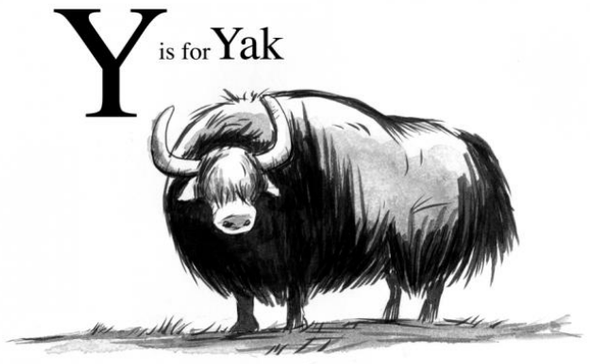
Fast Flow

Beat The Benchmark!

Burst round trip between 2 threads



Beware: Java Benchmarking!

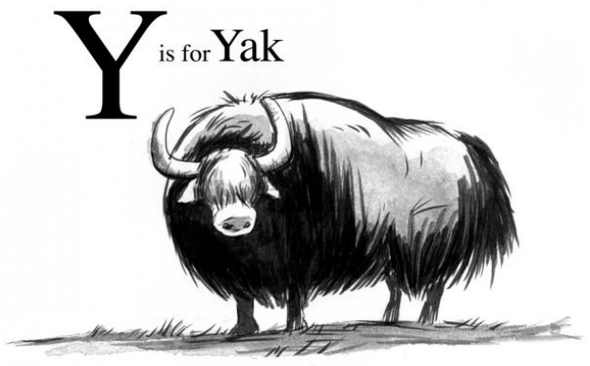


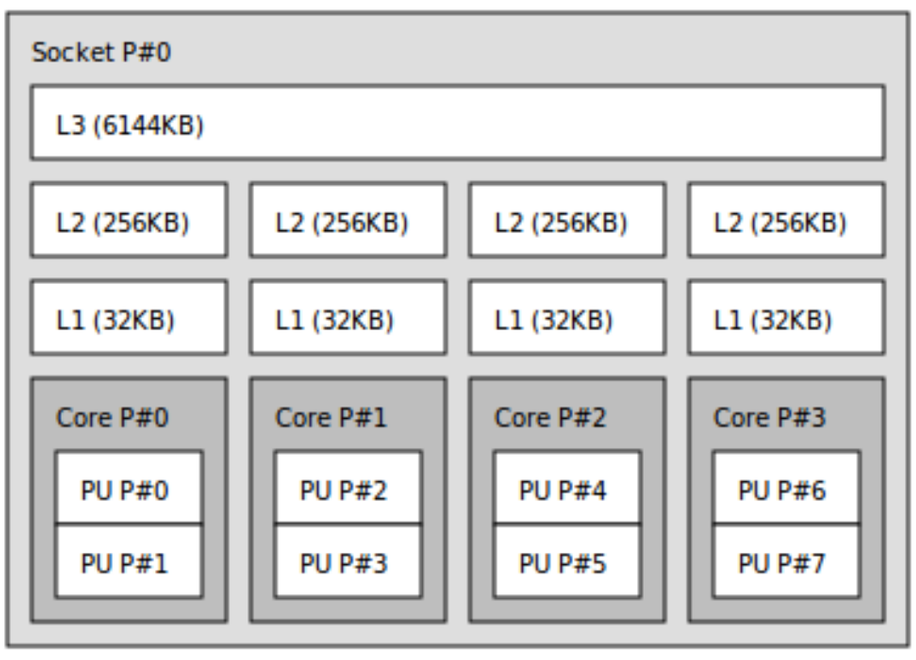
Use JMH!
Read the samples!

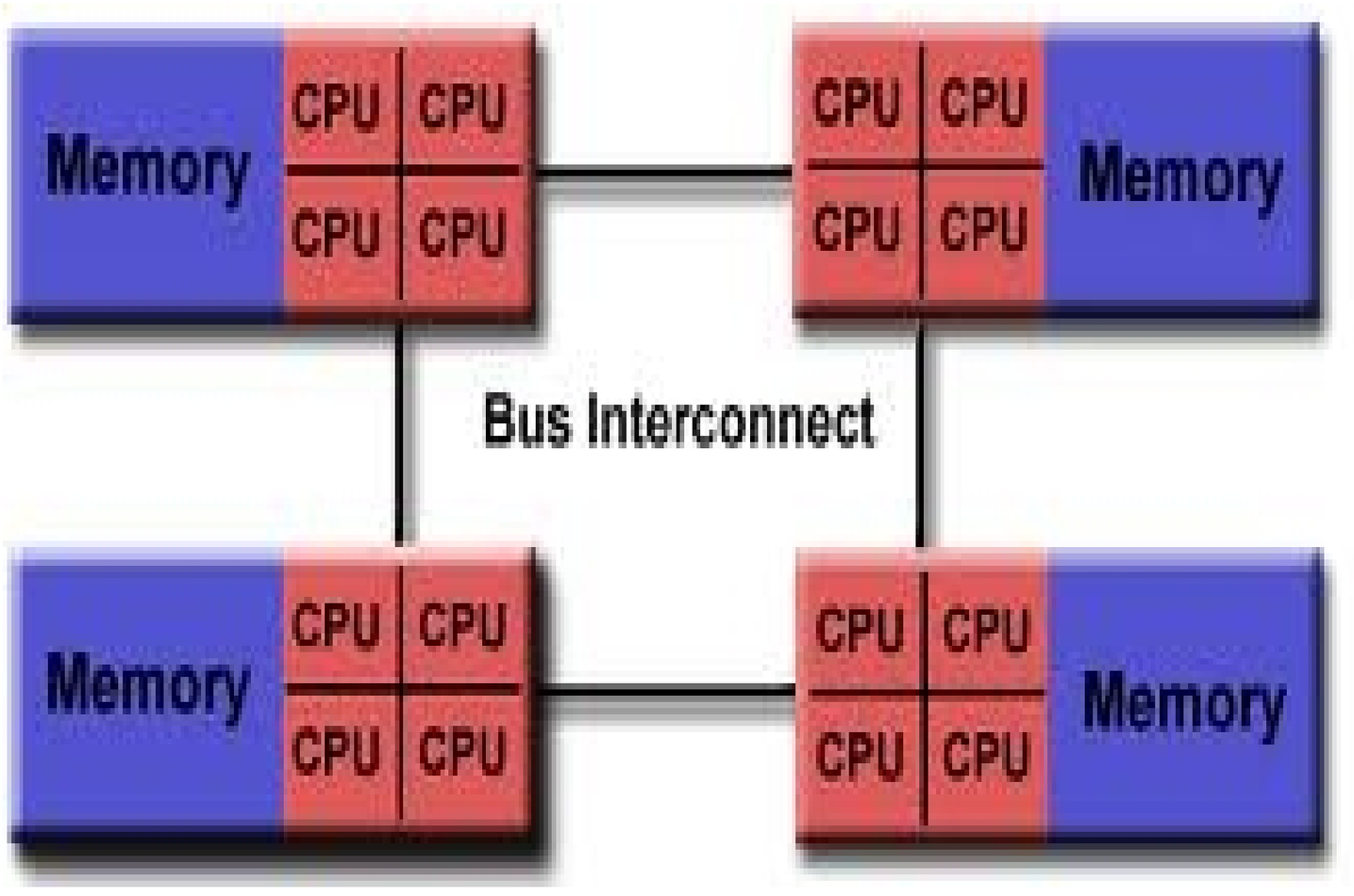
See:

- JMH project: http://openjdk.java.net/projects/code-tools/jmh/
- JMH introductory post: http://psy-lob-saw.blogspot.com/2013/04/writing-java-micro-benchmarks-with-jmh.html
- Benchmarking queue latency with JMH: http://psy-lob-saw.blogspot.com/2013/12/jaq-spsc-latency-benchmarks1.html

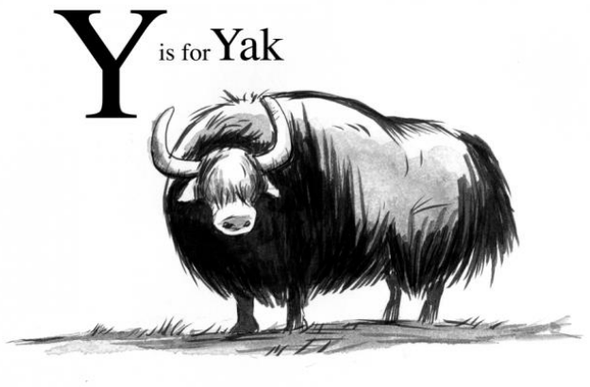
Thread Affinity

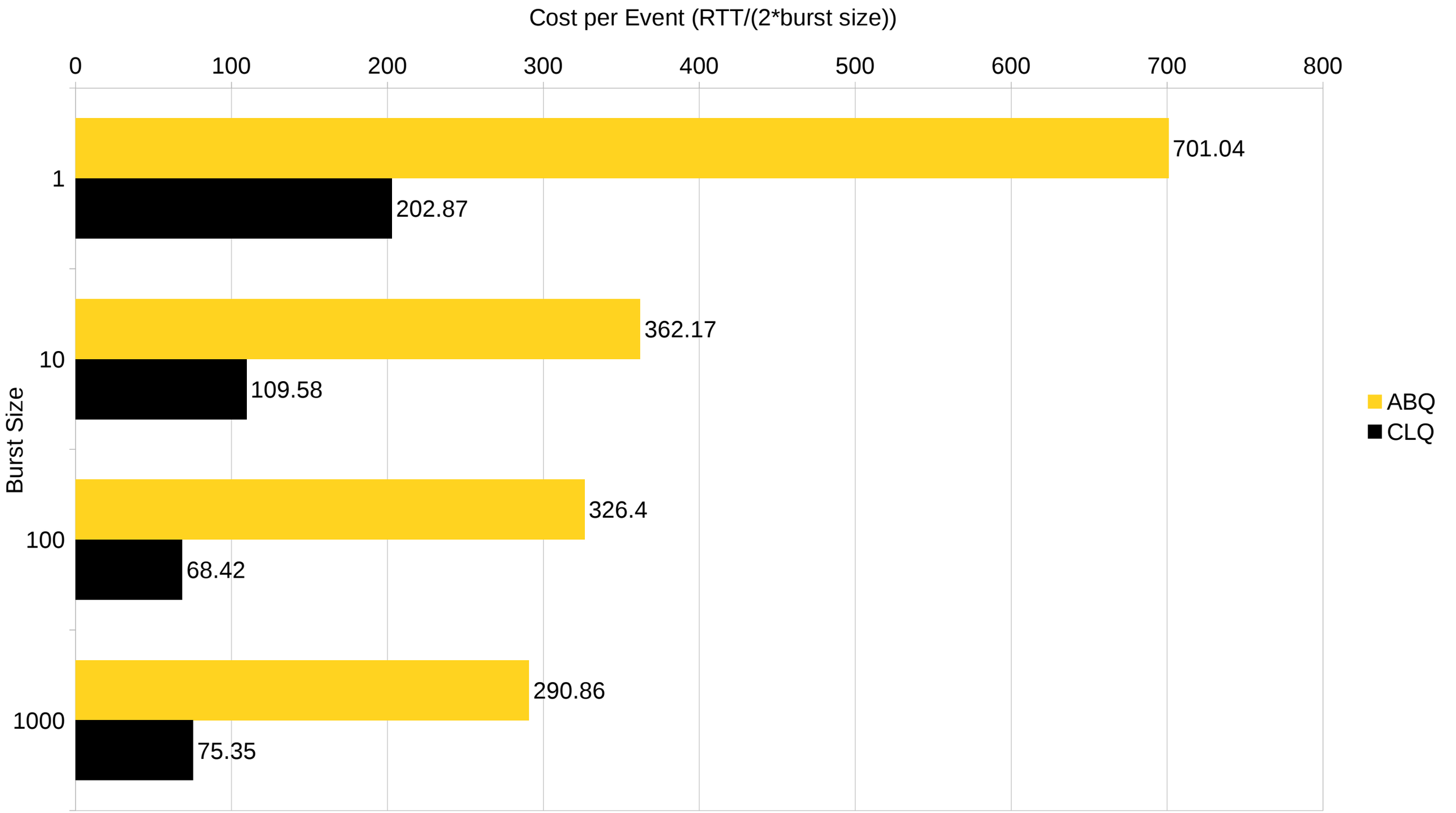




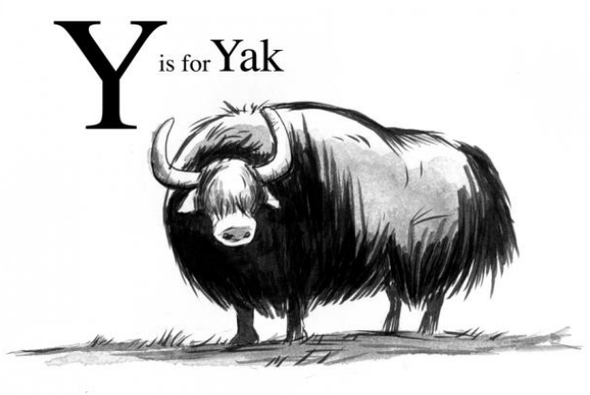


ArrayBlockingQueue vs. ConcurrentLinkedQueue





SPSC Bounded Wait Free Queue

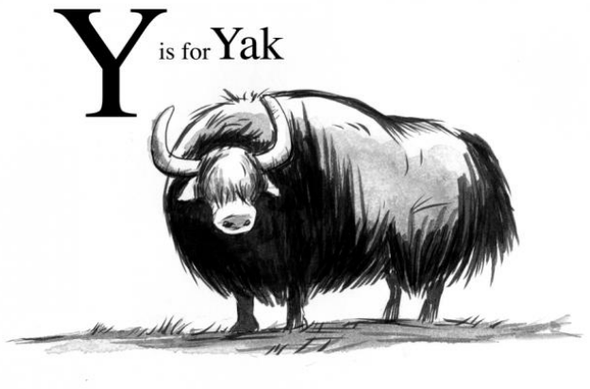


- **S**ingle **P**roducer **S**ingle **C**onsumer
- Java.util.Queue, **but only interested in offer/poll**
- Bounded - Of finite capacity
- Wait Free - Each thread will make progress

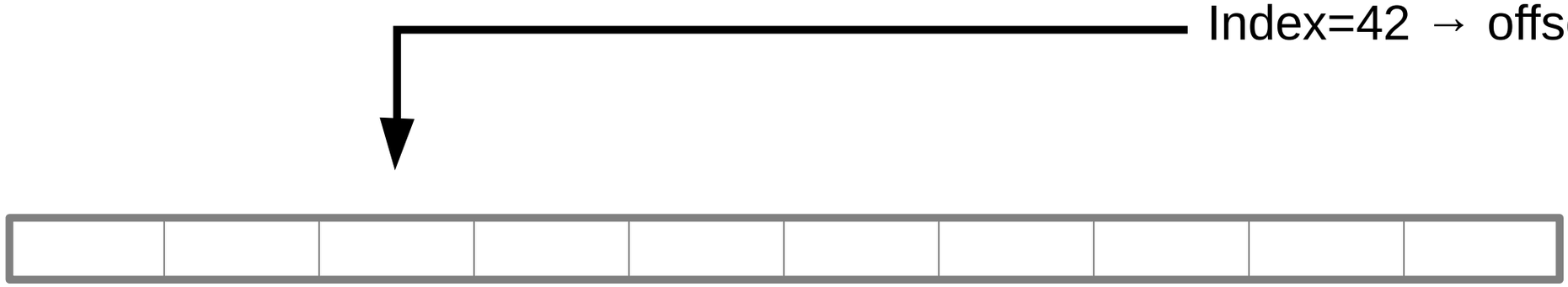
See:

- Lock Free vs Wait Free definition: www.1024cores.net/home/lock-free-algorithms/introduction

Circular Array



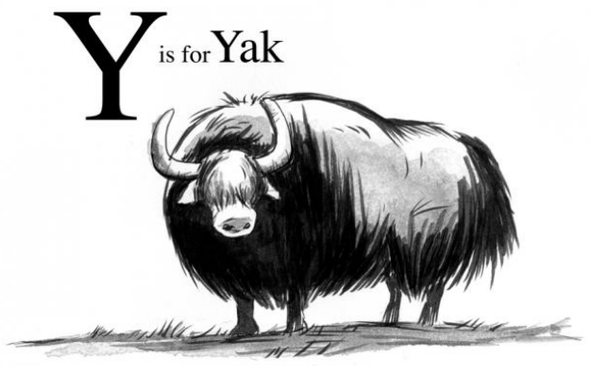
Index=42 → offset =2



E.g:
Capacity is 10
→ calcOffset(1) == 1
→ calcOffset(9) == 9
→ calcOffset(42) == 2

Usage:
int O = calcOffset(X);
spElement(O, E1);
→ lpElement(O) == E1

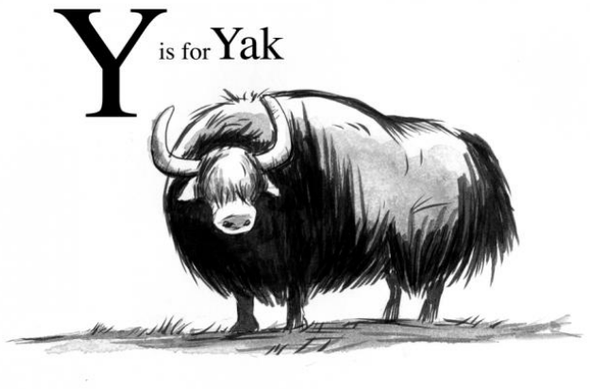
CircularArray1



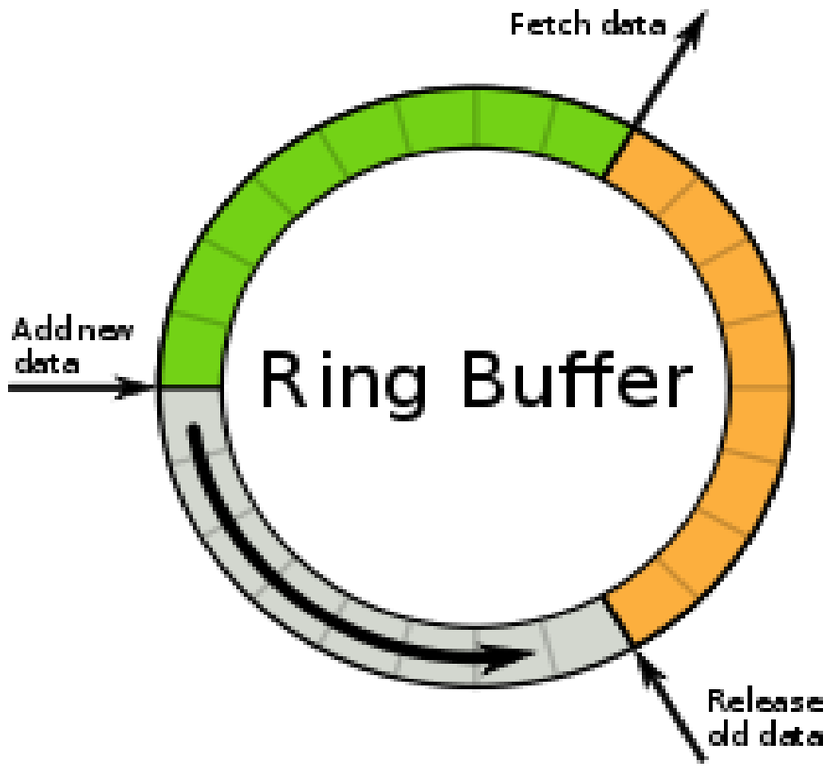

```
public abstract class CircularArrayQueue1<E> extends AbstractQueue<E> {
    private final E[] buffer;
    @SuppressWarnings("unchecked")
    public CircularArrayQueue1(int capacity) {
        buffer = (E[]) new Object[capacity];
    }
    protected final int calcOffset(final long index) {
        return (int) (index % buffer.length);
    }
    protected final void spElement(int offset, final E e) {
        buffer[offset] = e;
    }
    protected final E lpElement(final int offset) {
        return buffer[offset];
    }
    protected final int capacity() {
        return buffer.length;
    }
}
```

Yay! Code!

Circular Array Queue Mechanics



- Offer writes to the producerIndex, moving it counter clock wise
- Poll reads from the consumerIndex, chasing the producerIndex
- consumerIndex == producerIndex → Queue is empty
- consumerIndex + capacity == producerIndex → Queue is full



Lamport

producerIndex=42 → offset=2

E	E	null	null	null	null	null	null	null	E

consumerIndex=39 → offset=9

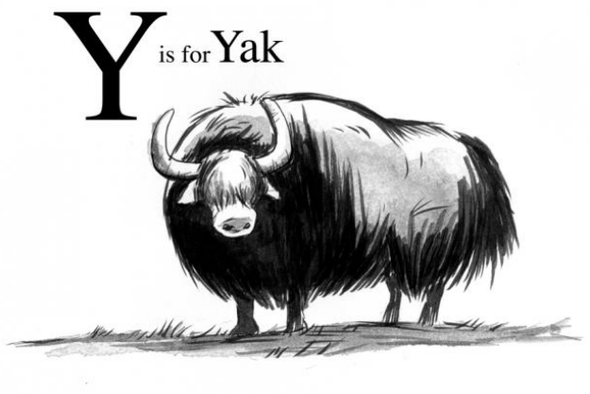
Offer/Poll look to peer counters to detect full/empty

If an index changes between reads → READ MISS!!!

See:

- L. Lamport. Concurrent reading and writing. Commun. ACM, 20(11):806-811, 1977.

Cache Misses



LLC

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
2	*	*	V2	*	*	*	*	*
3	*	*	*	*	*	*	V5	*
4	*	V4	*	*	*	*	*	*
5	*	*	V5	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	V7	*	*	*	*	*
8	*	V8	*	*	*	*	*	*
9	*	*	*	V9	*	*	*	*
10	*	*	V10	*	*	*	*	*
11	*	*	*	*	*	V11	*	*
12	*	*	*	*	*	*	V12	*

CPU 1 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
7	*	*	V7	*	*	*	*	*
12	*	*	*	*	*	*	V12	*

CPU 2 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
6	*	*	V6	*	*	*	*	*
10	*	*	*	*	*	*	V10	*

Cache Misses

LLC

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
2	*	*	V2	*	*	*	*	*
3	*	*	*	*	*	*	V5	*
4	*	V4	*	*	*	*	*	*
5	*	*	V5	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	V7	*	*	*	*	*
8	*	V8	*	*	*	*	*	*
9	*	*	*	V9	*	*	*	*
10	*	*	V10	*	*	*	*	*
11	*	*	*	*	*	V11	*	*
12	*	*	*	*	*	*	V12	*

CPU 1 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
7	*	*	V7	*	*	*	*	*
12	*	*	*	*	*	*	V12	*

CPU 2 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
6	*	*	V6	*	*	*	*	*
10	*	*	*	*	*	*	V10	*

Cache Misses

LLC

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
2	*	*	V2	*	*	*	*	*
3	*	*	*	*	*	*	V5	*
4	*	V4	*	*	*	*	*	*
5	*	*	V5	*	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	V7	*	*	*	*	*
8	*	V8	*	*	*	*	*	*
9	*	*	*	V9	*	*	*	*
10	*	*	V10	*	*	*	*	*
11	*	*	*	*	*	V11	*	*
12	*	*	*	*	*	*	V12	*

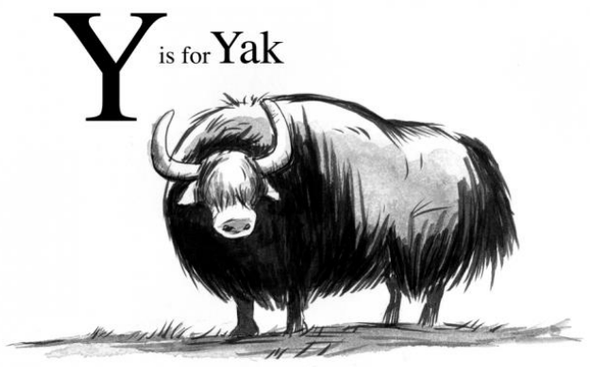
CPU 1 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
7	*	*	V7	*	*	*	*	*
12	*	*	*	*	*	*	V12	*

CPU 2 – L1

0	*	V0	V01	*	*	*	*	*
1	*	*	*	V1	*	*	*	*
6	*	*	V6	*	*	*	*	*
10	*	*	*	*	*	*	V10	*

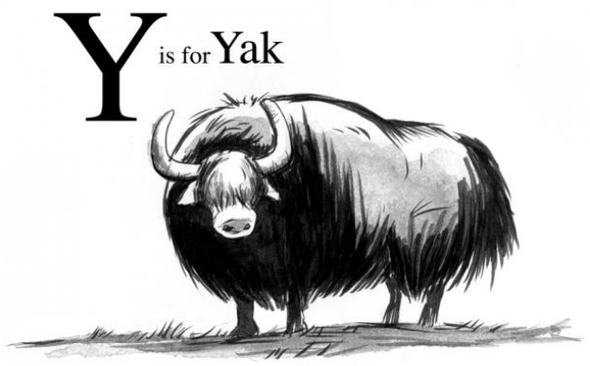
Cache Misses



- Read Miss
- Write Miss
- **A write invalidates all other copies!**

- Miss cost varies by distance:
  - `L1 CACHE hit,   ~4 cycles`
  - `L2 CACHE hit (L1 Miss), ~10`
  - `L3 CACHE hit (L2 Miss), ~40-75`

Thompson



producerIndex=42 → offset=2
consumerIndexCache=35 → offset=5

E	E	null	null	null	null	null	null	null	E

producerIndexCache=41 → offset=1
consumerIndex=39 → offset=9

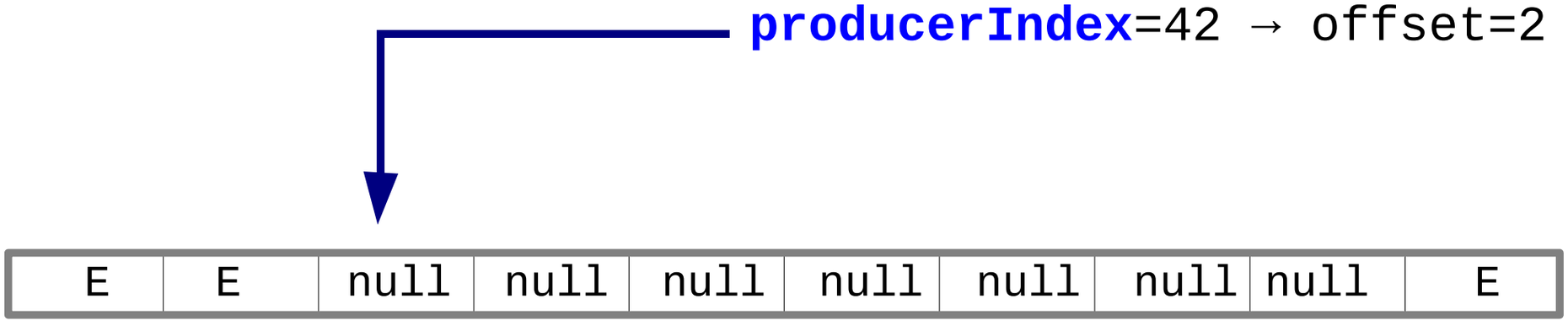
Offer/Poll look to peer counters to detect full/empty only when cached view exhausted

When local view is matched, read peer index, if it changed → READ MISS

See:

- Martin Thompson's presentation: http://www.infoq.com/presentations/Lock-Free-Algorithms

Fast Flow



Offer/Poll look to elements to detect queue full/empty!

Indexes are local to threads, only the array is shared data

See:

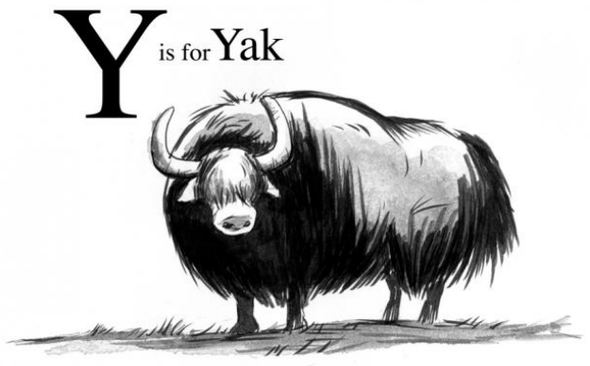
- Fast Flow SPSC paper: http://arxiv.org/pdf/1012.1824.pdf

Hungarian Notation
and related Memory Barriers

- V lp*() - load plain
- void sp*(V) - store plain
- V lv*() - load volatile → LoadLoad
- void so*(V) - store ordered → StoreStore
- void sv*(V) - store volatile → StoreLoad

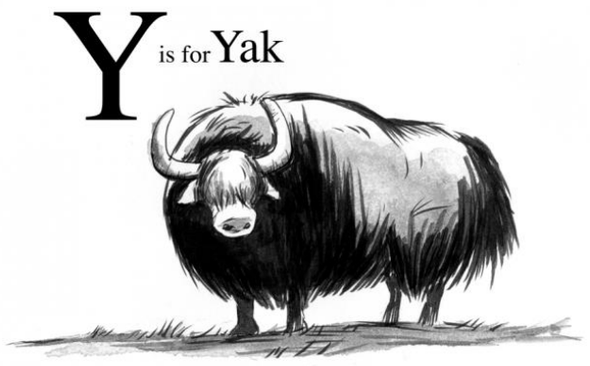
See:
- Pershing on Memory Barriers: http://preshing.com/20120710/memory-barriers-are-like-source-control-operations/

Lamport Queue 1




```
public final class LamportQueue1<E> extends CircularArrayQueue1<E> implements Queue<E> {
    private volatile long producerIndex = 0;
    private volatile long consumerIndex = 0;
    public LamportQueue1(final int capacity) {
        super(capacity);
    }

    private long lvProducerIndex() {
        return producerIndex; // LoadLoad
    }

    private void svProducerIndex(long producerIndex) {
        this.producerIndex = producerIndex; // StoreLoad
    }

    private long lvConsumerIndex() {
        return consumerIndex; // LoadLoad
    }

    private void svConsumerIndex(long consumerIndex) {
        this.consumerIndex = consumerIndex; // StoreLoad
    }
```

Lamport Queue 1




```
public boolean offer(final E e) {
    if (null == e) {
        throw new NullPointerException("Null is not a valid element");
    }

    final long currentProducerIndex = lvProducerIndex(); // LoadLoad
    final long wrapPoint = currentProducerIndex - capacity();
    if (lvConsumerIndex() <= wrapPoint) { // LoadLoad
        return false;
    }

    final int offset = calcOffset(currentProducerIndex);
    spElement(offset, e);
    svProducerIndex(currentProducerIndex + 1); // StoreLoad
    return true;
}
```

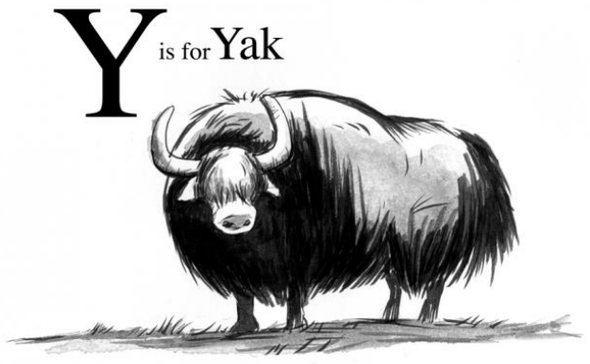
Lamport Queue 1


```
public E poll() {
    final long currentConsumerIndex = lvConsumerIndex(); // LoadLoad
    if (currentConsumerIndex >= lvProducerIndex()) { // LoadLoad
        return null;
    }

    final int offset = calcOffset(currentConsumerIndex);
    final E e = lpElement(offset);
    spElement(offset, null);
    svConsumerIndex(currentConsumerIndex + 1); // StoreLoad
    return e;
}
```

Memory Barriers: LoadLoad

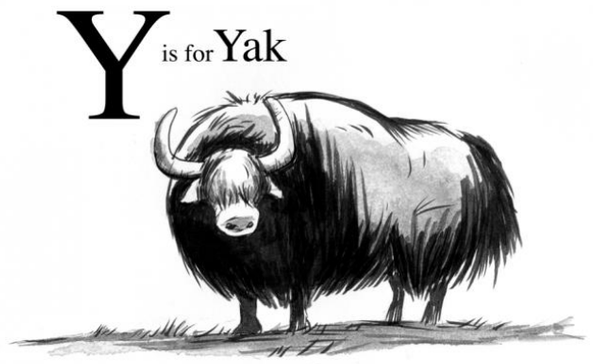
The sequence: Load1; **LoadLoad**; Load2 ensures that Load1's data are loaded before data accessed by Load2 and all subsequent load instructions are loaded.

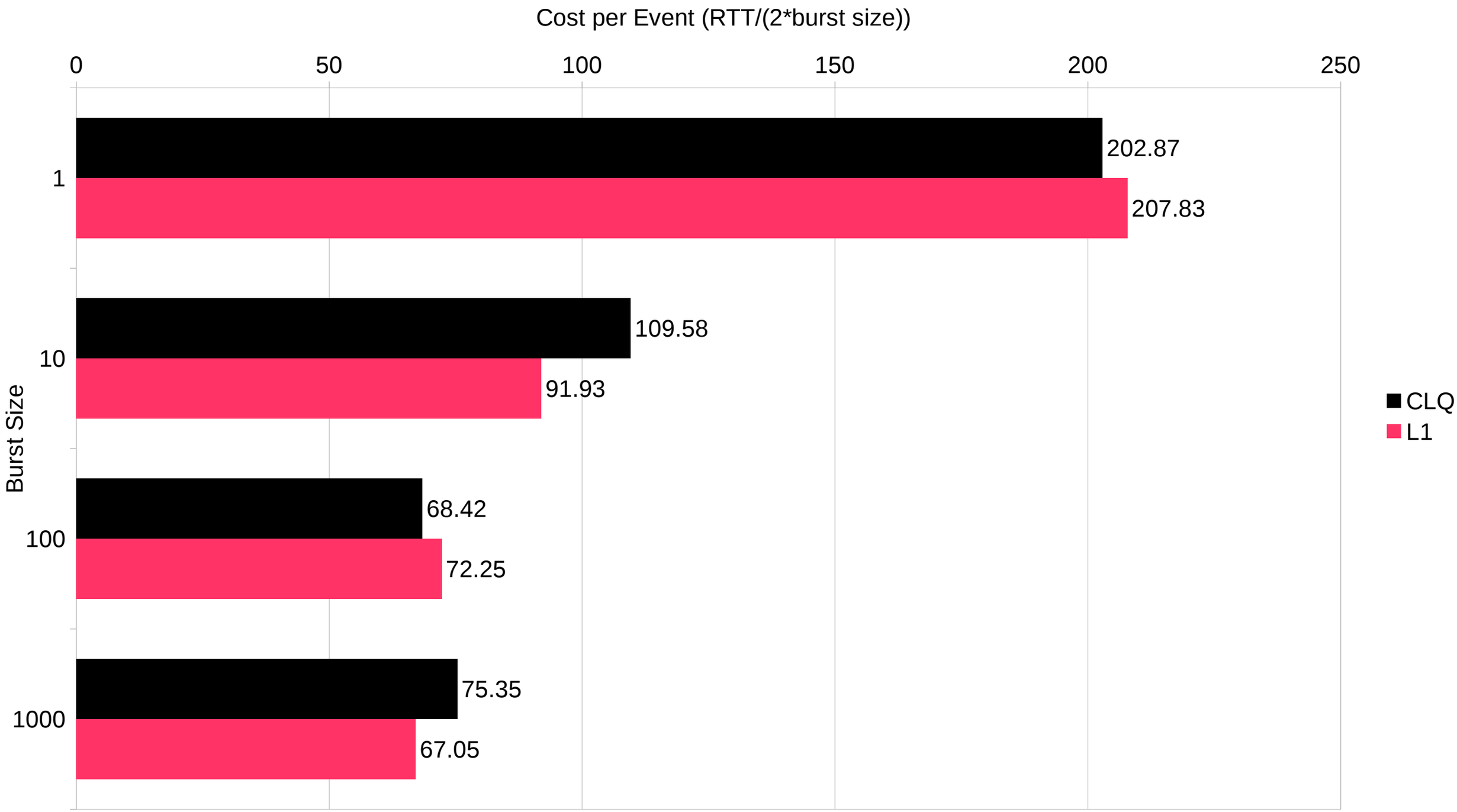
Memory Barriers: StoreLoad



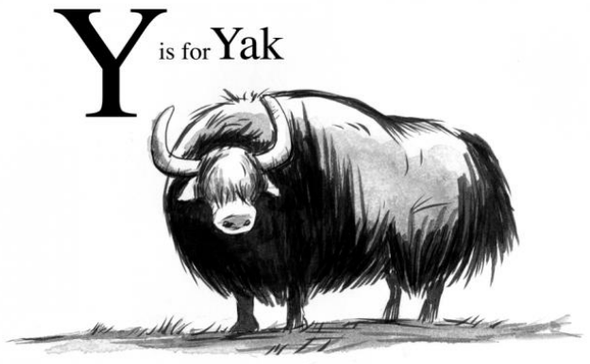
The sequence: Store1; **StoreLoad**; Load2 ensures that Store1's data are made visible to other processors (i.e., flushed to main memory) before data accessed by Load2 and all subsequent load instructions are loaded.

Lamport Queue 1



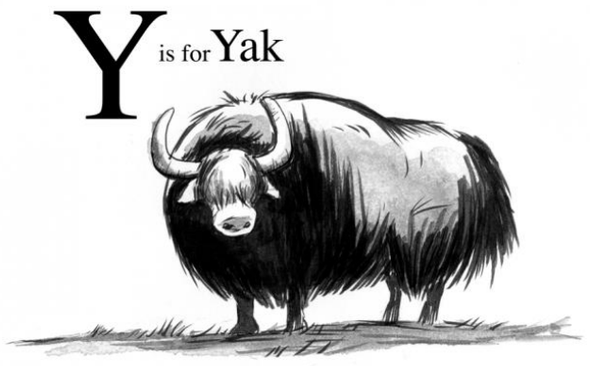


Instruction Level Optimizations



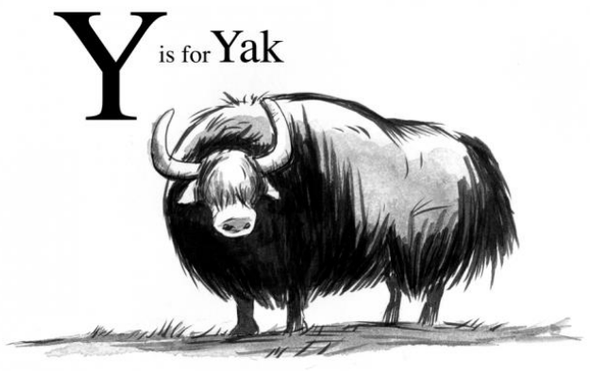
- Not all instructions are equal
- Find cheapest instructions...
- ...But still meet requirements

Lamport Queue 2: Cheaper HB



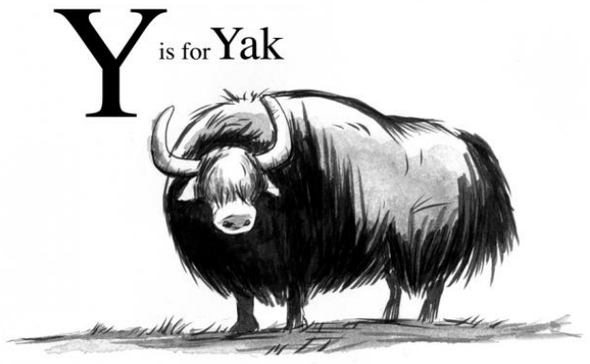

```
public final class LamportQueue2<E> extends CircularArrayQueue1<E> {
    private final AtomicLong producerIndex = new AtomicLong();
    private final AtomicLong consumerIndex = new AtomicLong();

    public LamportQueue2(final int capacity) {
        super(capacity);
    }

    private long lvProducerIndex() {
        return producerIndex.get();
    }

    private void soProducerIndex(long index) {
        producerIndex.lazySet(index);
    }

    private long lvConsumerIndex() {
        return consumerIndex.get();
    }

    private void soConsumerIndex(long index) {
        consumerIndex.lazySet(index);
    }
```

Lamport Queue 2: Cheaper HB



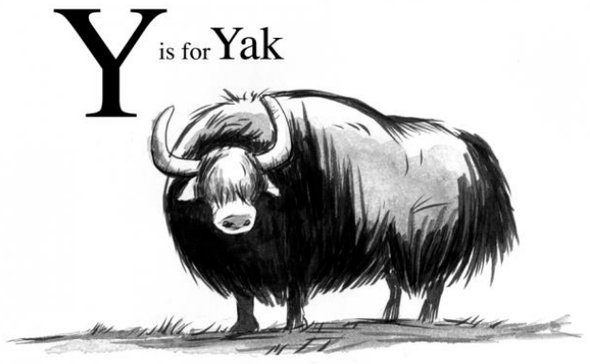

```
public boolean offer(final E e) {
    if (null == e) {
        throw new NullPointerException("Null is not a valid element");
    }

    final long currentProducerIndex = lvProducerIndex();// LoadLoad
    final long wrapPoint = currentProducerIndex - capacity();
    if (lvConsumerIndex() <= wrapPoint) { // LoadLoad
        return false;
    }

    final int offset = calcOffset(currentProducerIndex);
    spElement(offset, e);
    // was svProducerIndex(currentProducerIndex + 1); // StoreLoad
    soProducerIndex(currentProducerIndex + 1); // StoreStore
    return true;
}
```

Memory Barriers: StoreStore



The sequence: Store1; **StoreStore**; Store2 ensures that Store1's data are visible to other processors (i.e., flushed to memory) before the data associated with Store2 and all subsequent store instructions.

What is lazySet?

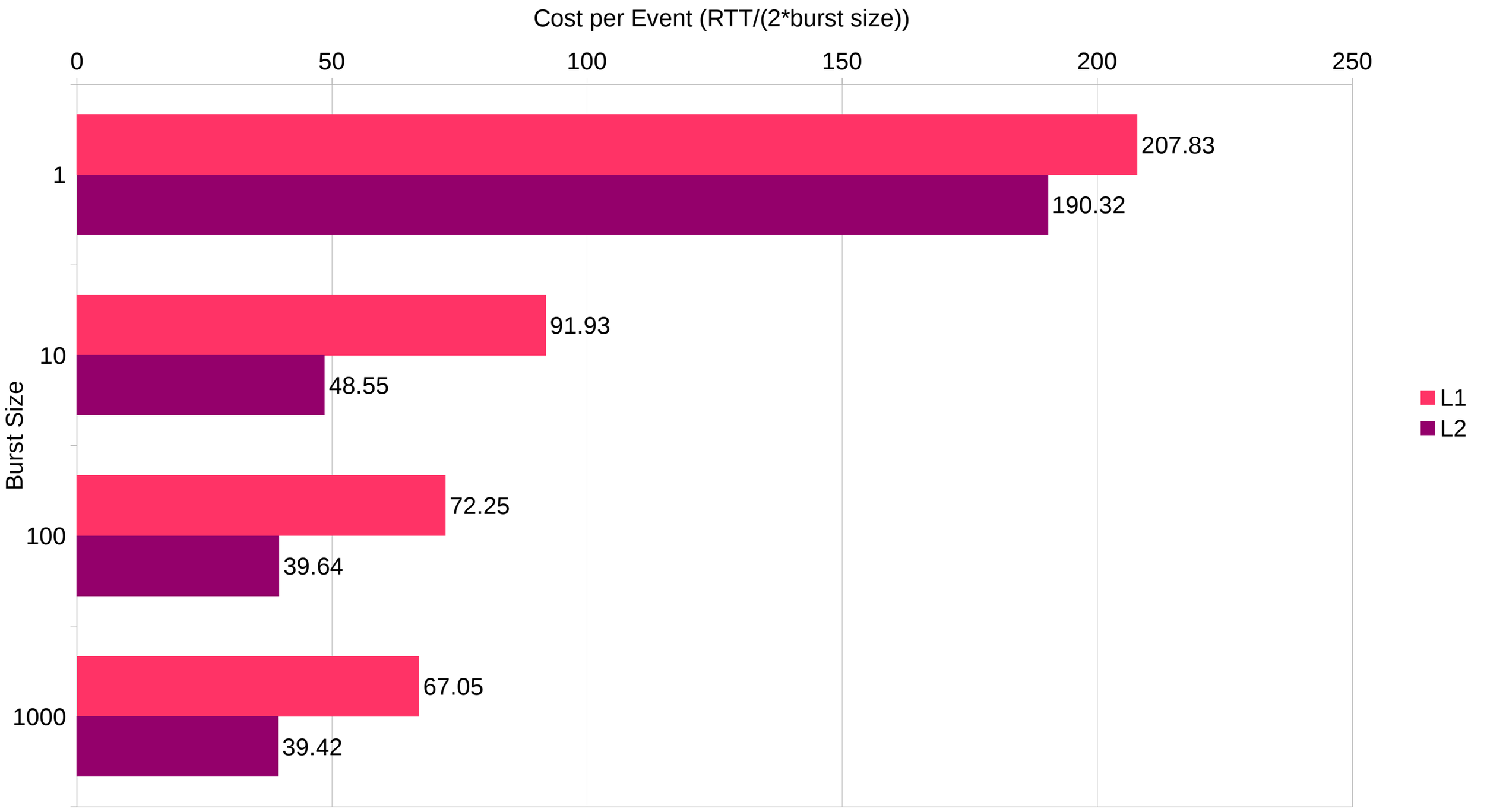


- AtomicLong.lazySet - A Doug Lea guarantee
- lazySet* == so*→ StoreStore
- Cheap, but not free

See:

- Post on lazySet origins and latency: http://psy-lob-saw.blogspot.com/2012/12/atomiclazyset-is-performance-win-for.html
- Original feature request: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6275329
- Concurrency interest discussion with Doug Lea: http://jsr166-concurrency.10961.n7.nabble.com/AtomicXXX-lazySet-and-happens-before-reasoning-td4512.html

Lamport Queue 2



Lamport Queue 3: Cheaper Modulo

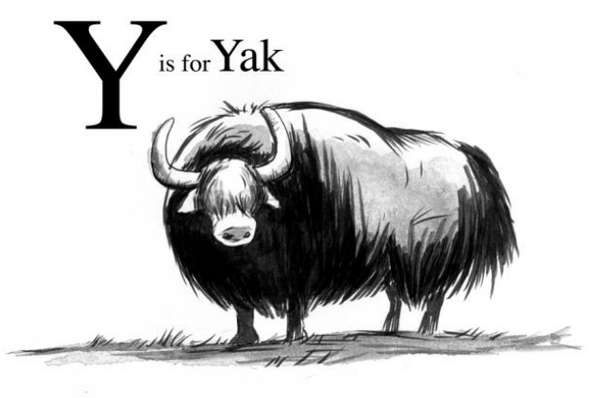
Using CircularArray2

- We make the array size a power of 2 because...
- If (isPowOf2(X)) → X % 4 == X & (4-1)
- Trade off space for improved performance

CircularArray2


```
public abstract class CircularArrayQueue2<E> extends AbstractQueue<E> {
    private final int capacity;
    private final int mask;
    private final E[] buffer;

    @SuppressWarnings("unchecked")
    public CircularArrayQueue2(int capacity) {
        this.capacity = Pow2.findNextPositivePowerOfTwo(capacity);
        mask = capacity() - 1;
        buffer = (E[]) new Object[capacity];
    }

    protected final void spElement(int offset, final E e) {
        buffer[offset] = e;
    }

    protected final E lpElement(final int offset) {
        return buffer[offset];
    }

    protected final int calcOffset(final long index) {
        // was: return (int) (index % buffer.length);
        return ((int) index) & mask;
    }

    protected final int capacity() {
        // was: return buffer.length
        return capacity;
    }
}
```

The problem with %



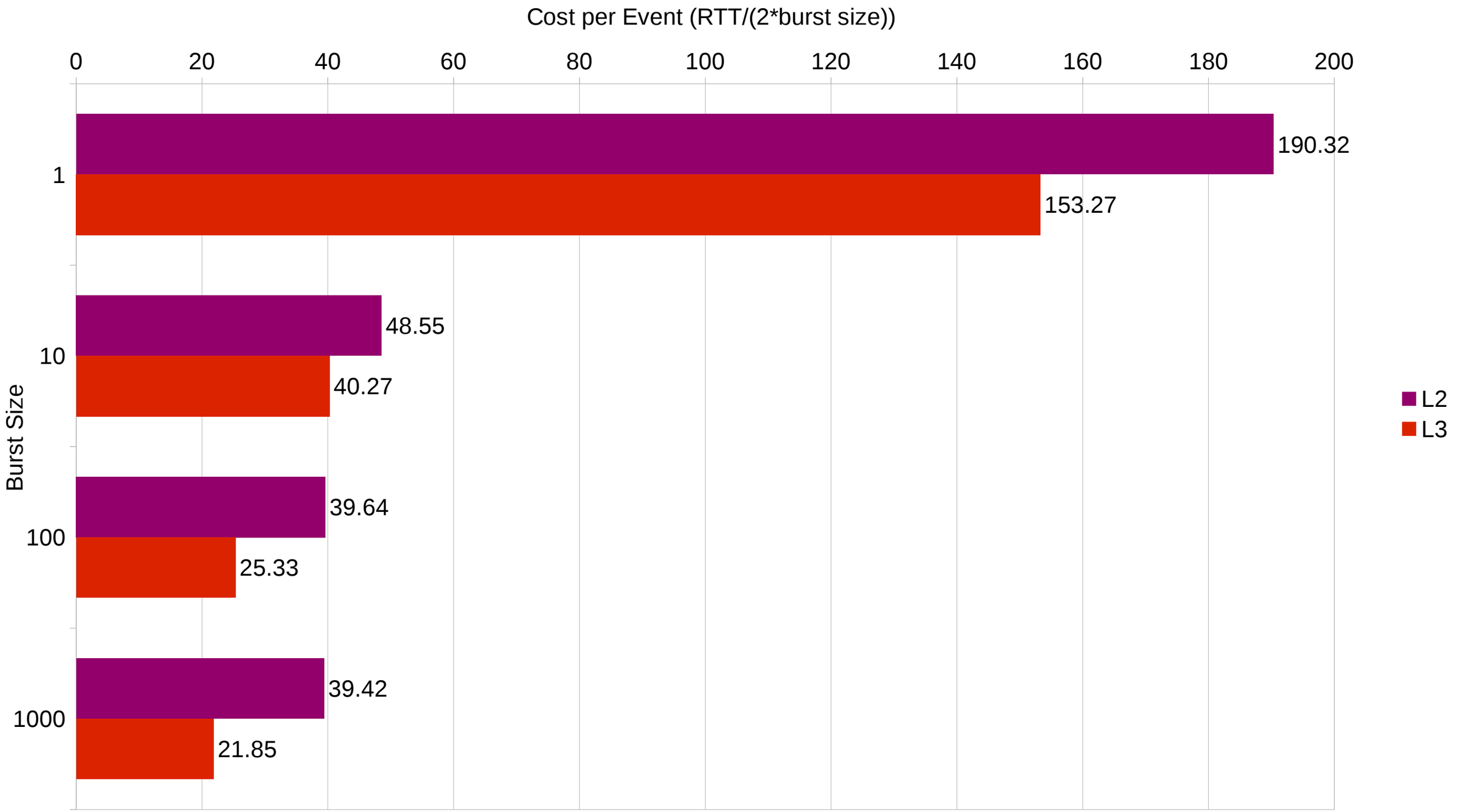
- Different instructions have different costs
- Some instructions have a **throughput impact**
- Consult Intel manuals or Agner Fog
- “+-&|...” **<** “*...” **<** “/%”

Instruction	Operands	µops fused domain	µops unfused domain	µops each port	Latency	Recipro-cal through put	Comments
DIV	r8	9	9	p0 p1 p5 p6	22-25	9	
DIV	r16	11	11	p0 p1 p5 p6	23-26	9	
DIV	r32	10	10	p0 p1 p5 p6	22-29	9-11	
DIV	r64	36	36	p0 p1 p5 p6	32-96	21-74	
AND OR XOR	r,r/i	1	1	p0156	1	0.25	
AND OR XOR	r,m	1	2	p0156 p23		0.5	
AND OR XOR	m,r/i	2	4	2p0156 2p237 p4	6	1	

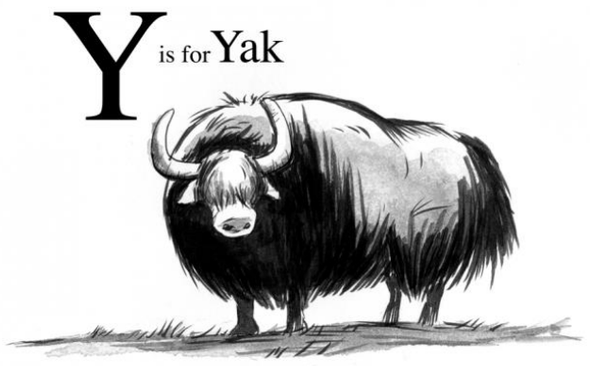
See:

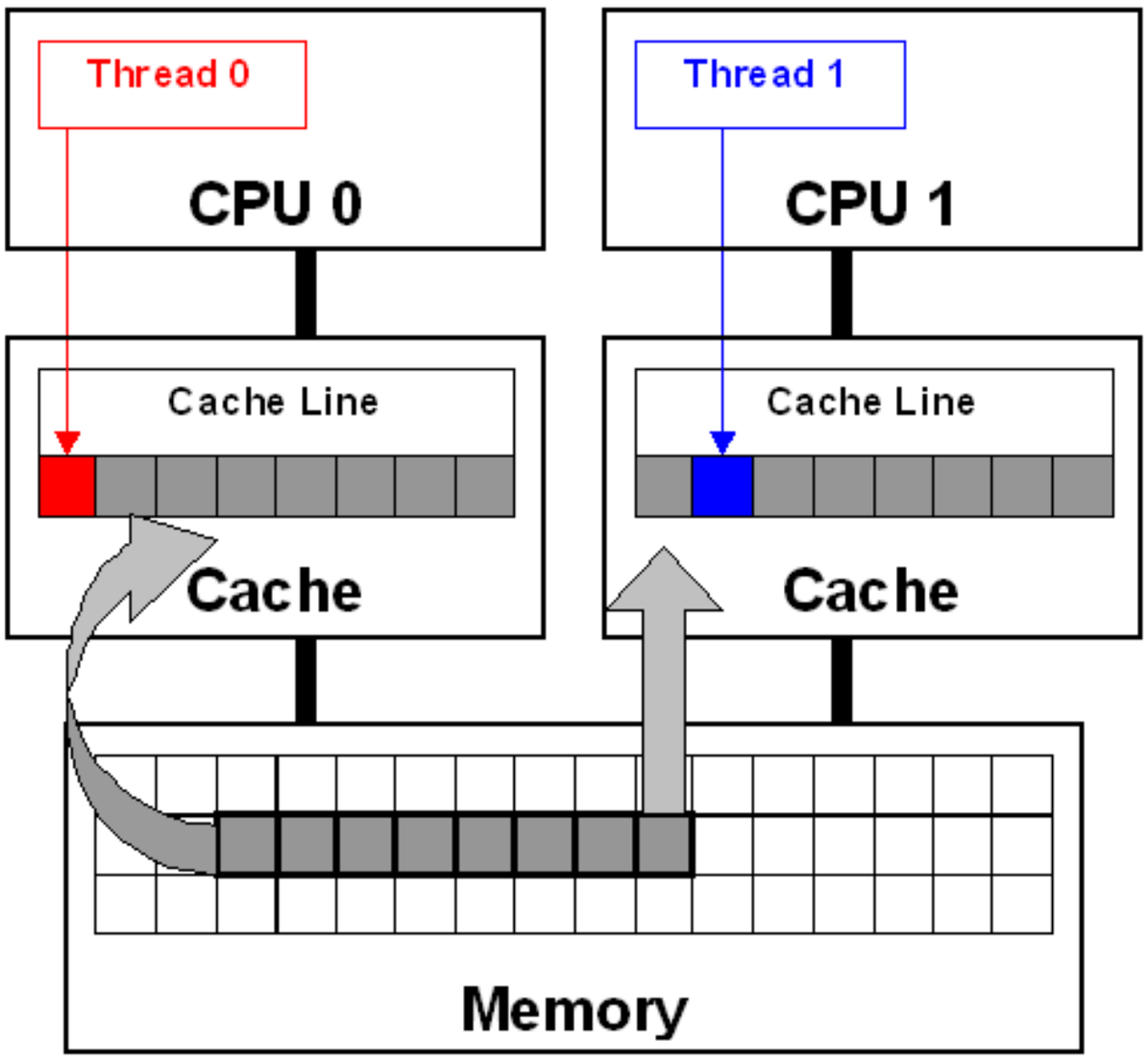
- Intel manuals: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html
- Agner Fog optimization manuals: http://www.agner.org/optimize/

Lamport Queue 3

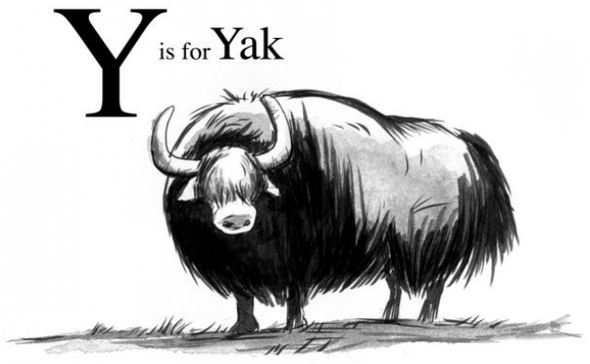


False Sharing





Lamport Queue 4: False Sharing



- 2 independent values share the same line
- The cache line is **usually** 64 bytes
- What's my Java object memory layout?

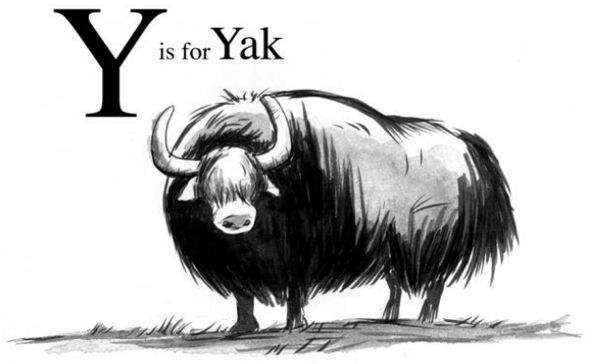
```
                LamportQueue1
OFFSET  SIZE      TYPE DESCRIPTION
     0    12           (object header)
    12     4 Object[] buffer
    16     8      long producerIndex
    24     8      long consumerIndex
```

```
                LamportQueue3
OFFSET  SIZE       TYPE DESCRIPTION
     0    12            (object header)
    12     4         int capacity
    16     4         int mask
    20     4   Object[] buffer
    24     4 AtomicLong producerIndex
    28     4 AtomicLong consumerIndex
```

See:

- Post on object memory layout: http://psy-lob-saw.blogspot.com/2013/05/know-thy-java-object-memory-layout.html
- JOL – The Java Object Layout tool used to get the above data: http://openjdk.java.net/projects/code-tools/jol
- Post on cache coherency: http://psy-lob-saw.blogspot.com/2013/09/diving-deeper-into-cache-coherency.html

Warning: sun.misc.Unsafe



- The underbelly of the JVM, not public API
- put*Ordered(ref,offset,value) == lazySet
- Use at your own risk, exercise extreme caution

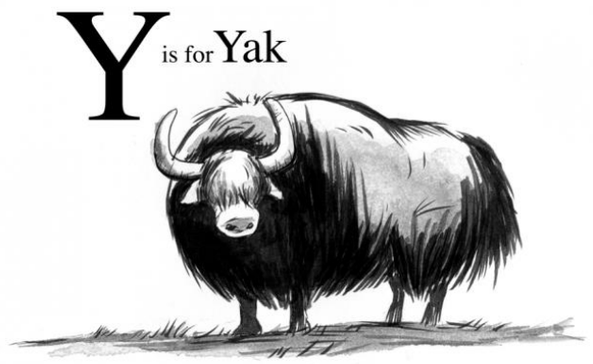
False Sharing - I

- Pad before AND after counter fields

```
abstract class VolatileLongCellPrePad{long p0,p1,p2,p3,p4,p5,p6;}
abstract class VolatileLongCellValue extends VolatileLongCellPrePad {
    protected volatile long value;
}
public final class VolatileLongCell extends VolatileLongCellValue {
    long p10,p11,p12,p13,p14,p15,p16;
    private final static long VALUE_OFFSET;
    static {
        try {
            VALUE_OFFSET = UNSAFE.objectFieldOffset(VolatileLongCellValue.class.getDeclaredField("value"));
        } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
        }
    }
    public VolatileLongCell(){
        this(0L);
    }
    public VolatileLongCell(long v){
        lazySet(v);
    }
    public void lazySet(long v) {
        UNSAFE.putOrderedLong(this, VALUE_OFFSET, v);
    }
    public void set(long v){
        this.value = v;
    }
    public long get(){
        return this.value;
    }
}
```

```
          VolatileLongCell
OFFSET  SIZE  TYPE DESCRIPTION
     0    12       (object header)
    12     4       (alignment/padding gap)
    16    56  long PADDING
    72     8  long value
    80    56  long PADDING
```

False Sharing - II



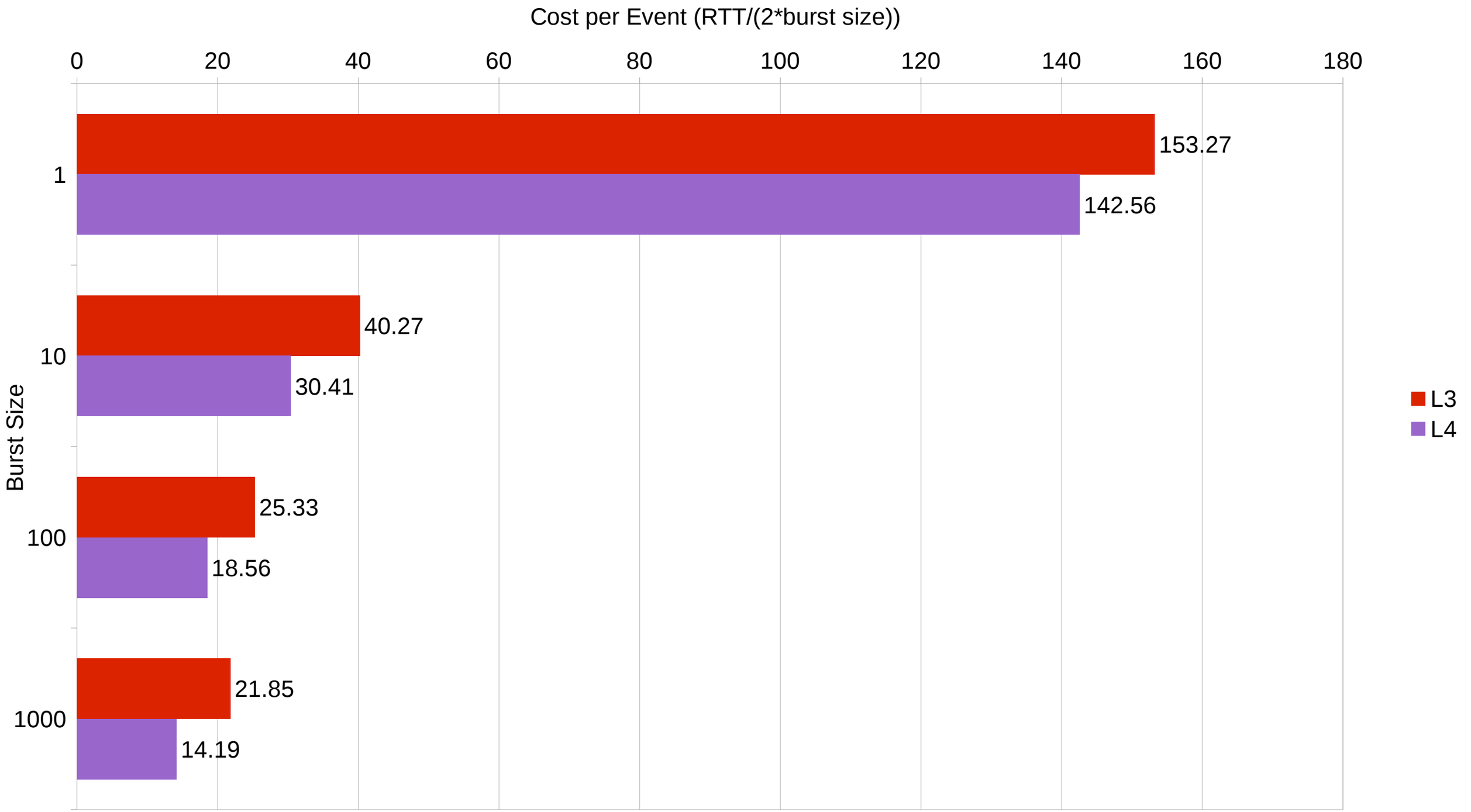
- Pad before AND after queue fields

```
                    LampportQueue4
    OFFSET  SIZE                TYPE DESCRIPTION
         0    12                     (object header)
        12     4                     (alignment/padding gap)
     16-144   128                long PADDING
       144     4                 int capacity
       148     4                 int mask
       152     4          Object[] buffer
       156     4 VolatileLongCell producerIndex
       160     4 VolatileLongCell consumerIndex
       164     4                     (alignment/padding gap)
    168-296   128                long PADDING
```

More False Sharing?

There are 3 more cases of false sharing ... but let's move on

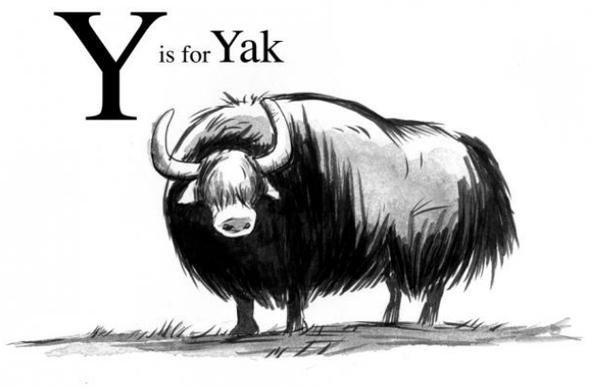
Lamport Queue 4

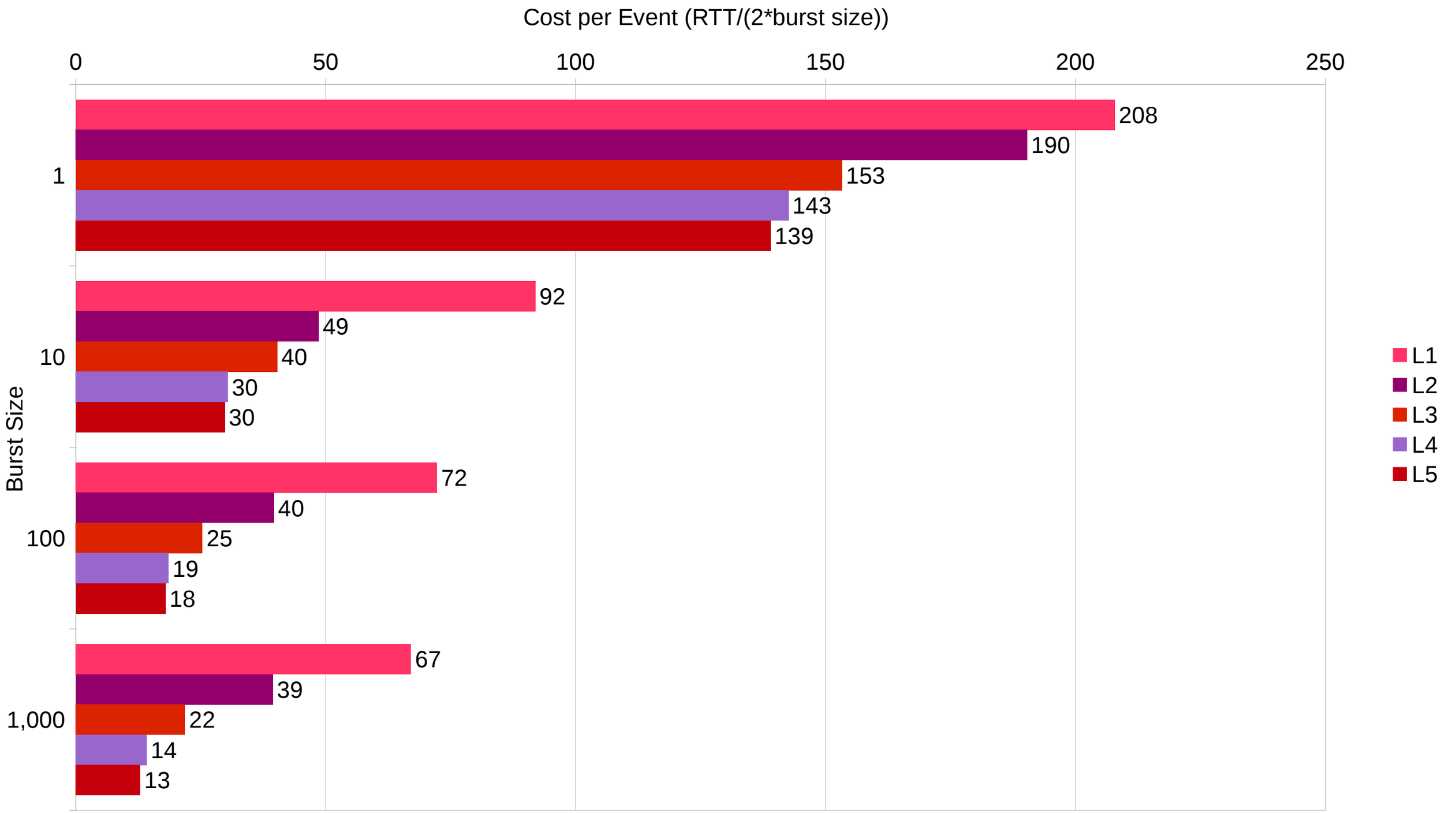


A few more tweaks...

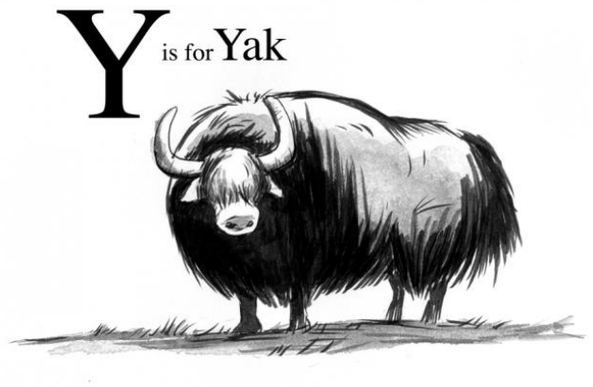
- Inline the counters
- Use Unsafe for array access

Lamport Queue 5





Thompson



producerIndex=42 → offset=2
consumerIndexCache=35 → offset=5

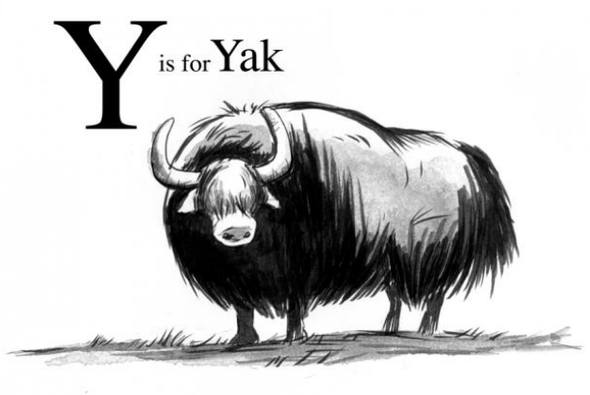
E	E	null	null	null	null	null	null	null	E

producerIndexCache=41 → offset=1
consumerIndex=39 → offset=9

Offer/Poll look to peer counters to detect full/empty only when cached view exhausted

When local view is matched, read peer index, if it changed → READ MISS

Fast Flow



producerIndex=42 → offset=2

E	E	null	null	null	null	null	null	null	E

consumerIndex=39 → offset=9

Offer/Poll look to elements to detect queue full/empty!

Indexes are local to threads, only the array is shared data

All together now