



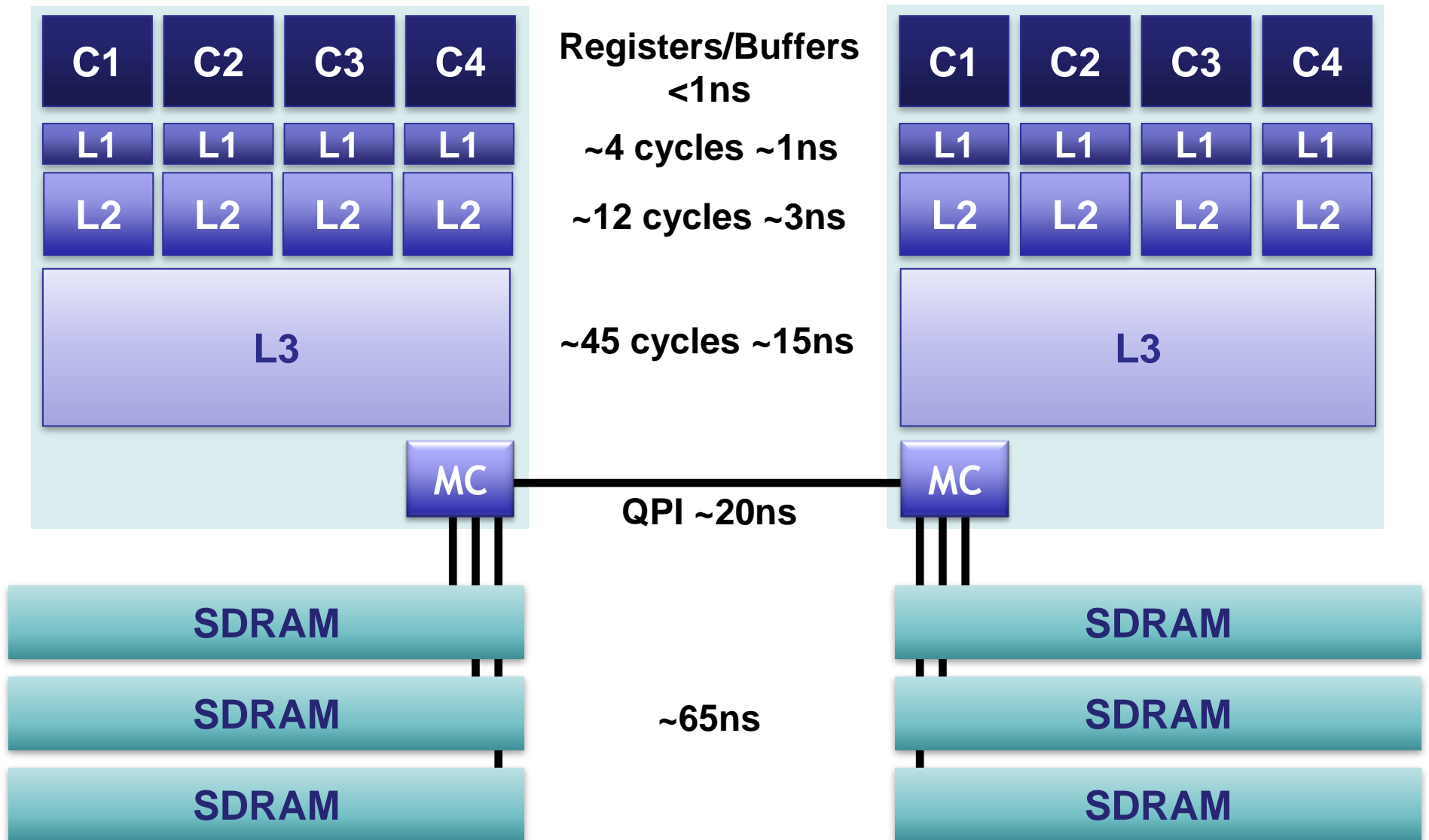
# ***Lock-Free Algorithms***

**Martin Thompson - @mjpt777**

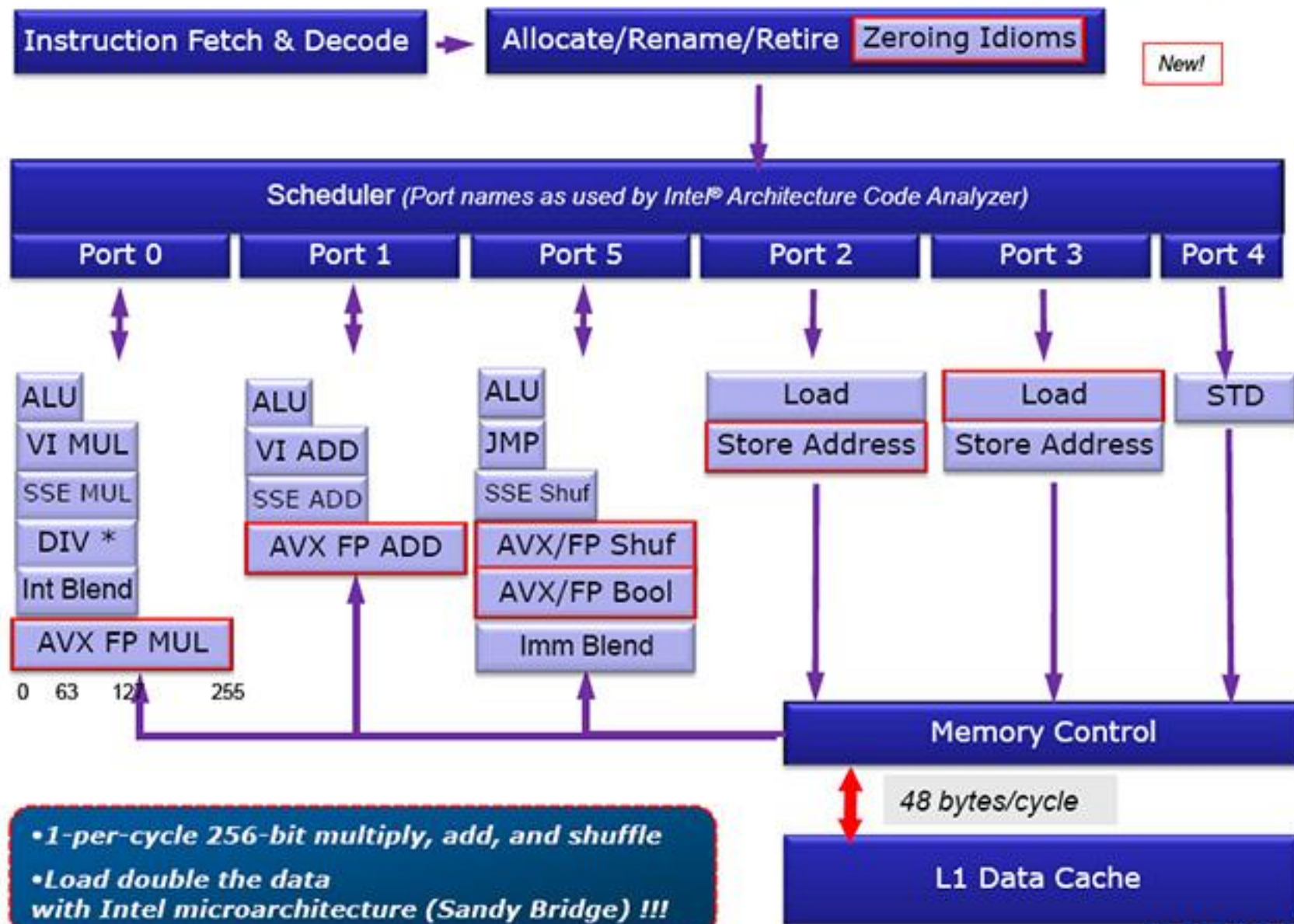
**Mike Barker - @mikeb2701**

# Modern Hardware

# Modern Hardware (Intel Nehalem)

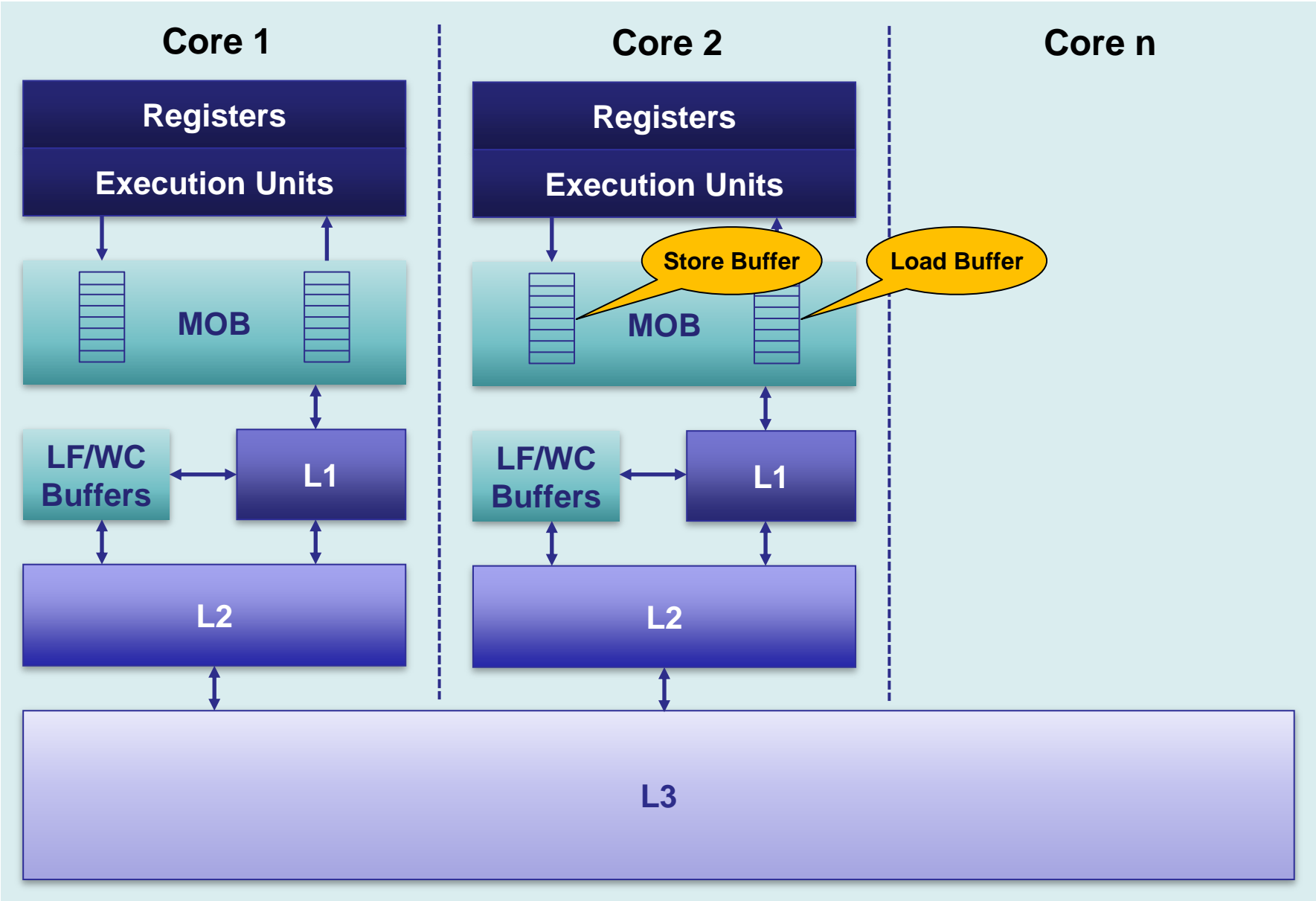


# Intel® Microarchitecture (Sandy Bridge) Highlights

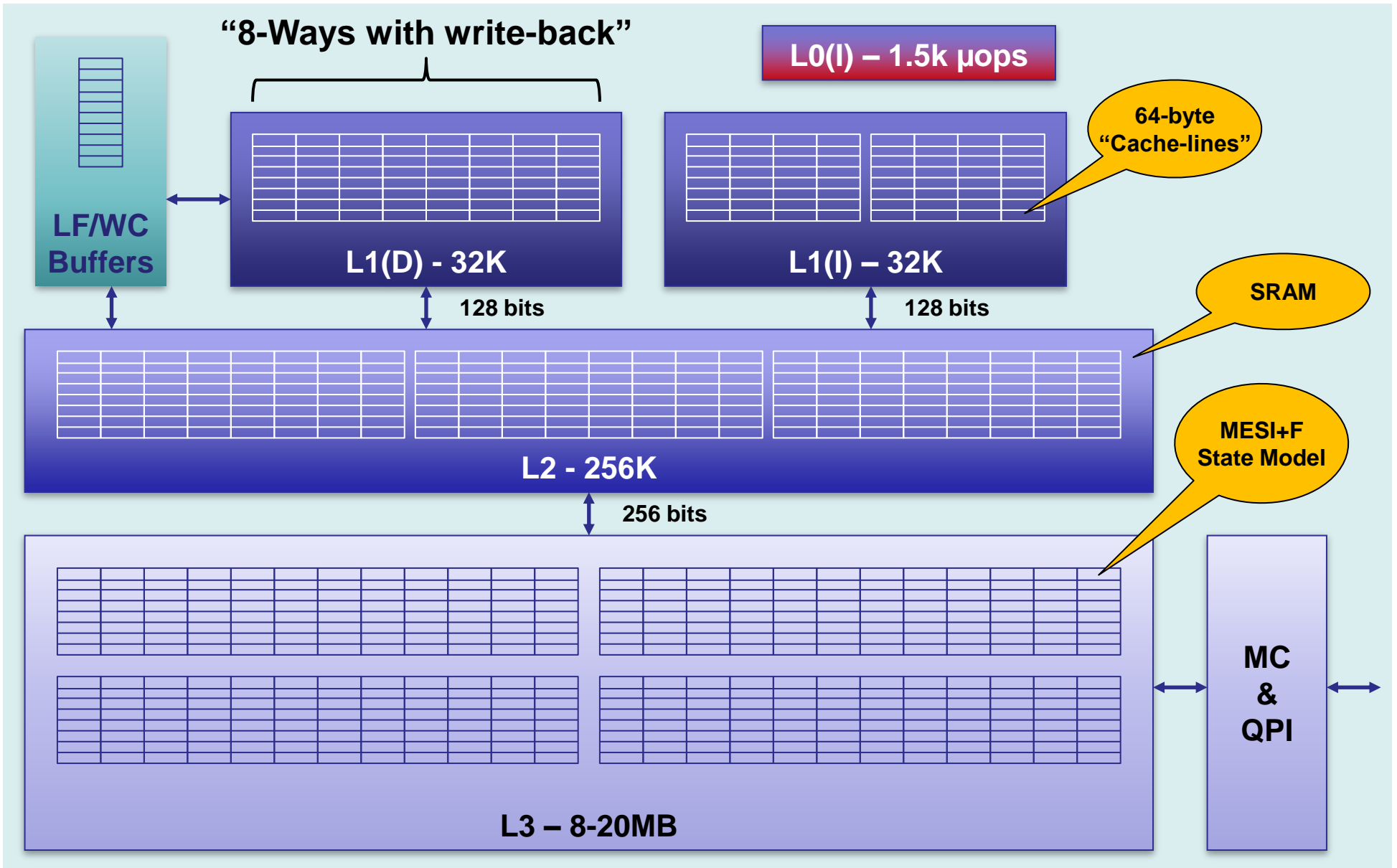


\* Not fully pipelined

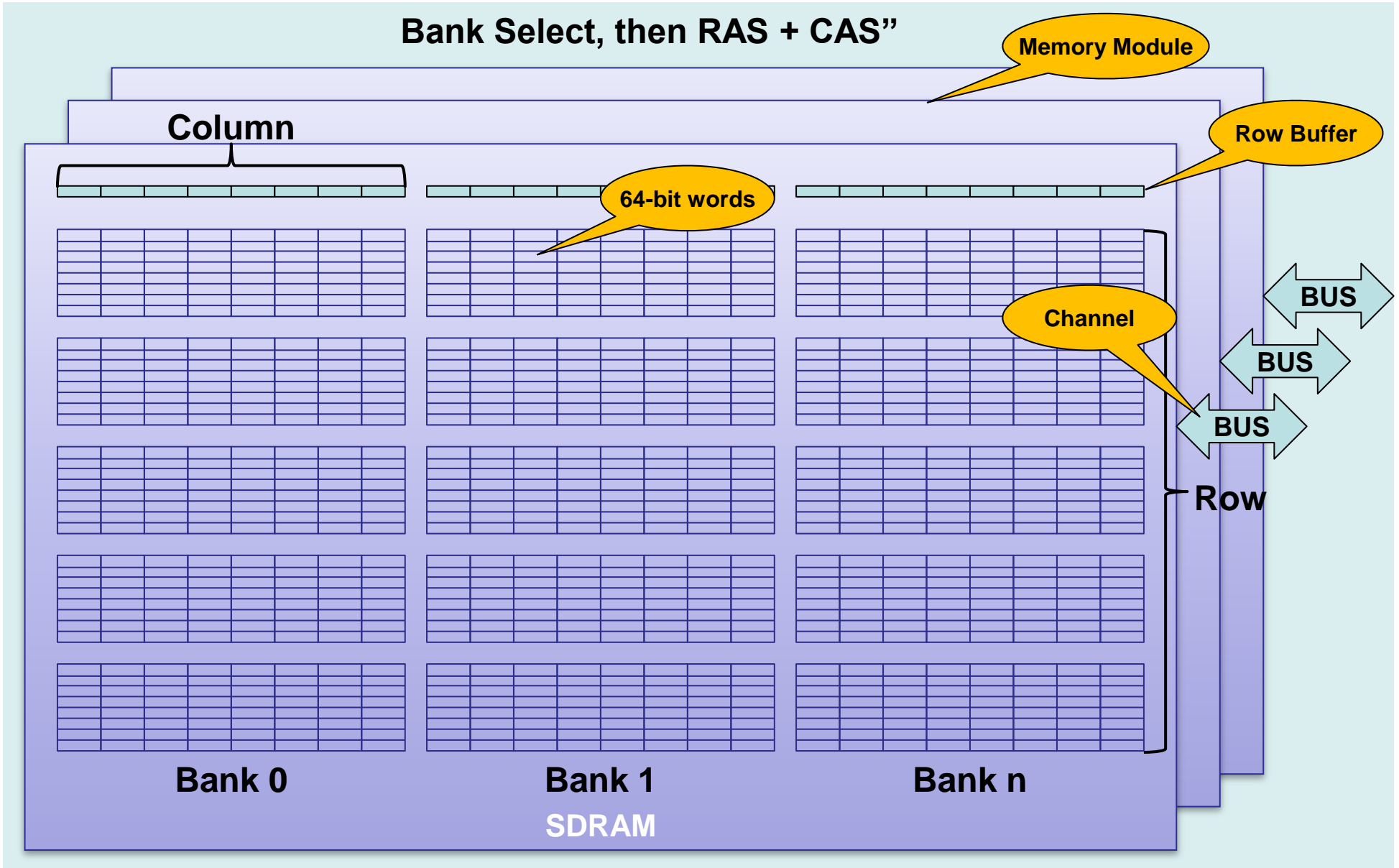
# Memory Ordering



# Cache Structure & Coherence



# Main Memory



# Memory Models



# Hardware Memory Models

Memory consistency models describe how threads may interact through shared memory consistently.

- Program Order (PO) for a single thread
- Sequential Consistency (SO) [Lamport 1979]
  - > What you expect a program to do! (for race free)
- Strict Consistency (Linearizability)
  - > Some special instructions
- Total Store Order (TSO)
  - > Sparc model that is stronger than SC
- x86/64 is SC + (Total Lock Order & Causal Consistency)
  - > <http://www.youtube.com/watch?v=WUfvvFD5tAA>
- Other Processors have weaker models

# Intel x86/64 Memory Model

<http://www.multicoreinfo.com/research/papers/2008/damp08-intel64.pdf>

<http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html>

- 1. Loads are not reordered with other loads.**
- 2. Stores are not reordered with other stores.**
- 3. Stores are not reordered with older loads.**
- 4. Loads may be reordered with older stores to different locations but not with older stores to the same location.**
- 5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).**
- 6. In a multiprocessor system, stores to the same location have a total order.**
- 7. In a multiprocessor system, locked instructions have a total order.**
- 8. Loads and stores are not reordered with locked instructions.**

# Language/Runtime Memory Models

**Some languages/Runtimes have a well defined memory model for portability:**

- **Java Memory Model (Java 5)**
- **C++ 11**
- **Erlang**

**For most other languages we are at the mercy of the compiler**

- **Instruction reordering**
- **C “volatile” is inadequate**
- **Register allocation for caching values**
- **No mapping to the hardware memory model**
- **Fences/Barriers need to be applied**

# **Measuring What Is Going On**

# Model Specific Registers (MSR)

- **Many and varied uses**
  - > **Timestamp Invariant Counter**
  - > **Memory Type Range Registers**
- **Performance Counters!!!**
  - > **L2/L3 Cache Hits/Misses**
  - > **TLB Hits/Misses**
  - > **QPI Transfer Rates**
  - > **Instruction and Cycle Counts**
  - > **Lots of others....**

## Accessing MSRs

```
void rdmsr(uint32_t msr, uint32_t* lo, uint32_t* hi)
{
    asm volatile("rdmsr" : "=a" lo, "=d" hi : "c" msr);
}
```

```
void wrmsr(uint32_t msr, uint32_t lo, uint32_t hi)
{
    asm volatile("wrmsr" :: "c" msr, "a" lo, "d" hi);
}
```

## On Linux

```
f = new RandomAccessFile("/dev/cpu/0/msr", "rw");  
ch = f.getChannel();  
buffer.order(ByteOrder.LITTLE_ENDIAN);  
ch.read(buffer, msrNumber);  
long value = buffer.getLong(0);
```

**Contention Is The  
Enemy**



# Contention

- **Managing Contention**
  - > Locks
  - > CAS Techniques
- **Little's & Amdahl's Laws**
  - >  $L = \lambda W$
  - > *Sequential Component Constraint*



- **Single Writer Principle**
- **Shared Nothing Designs**

# Locks

# Software Locks

- **Mutex, Semaphore, Critical Section, etc.**
  - > **What happens when un-contended?**
  - > **What happens when contention occurs?**
  - > **What if we need condition variables?**
  - > **What are the cost of software locks?**
  - > **Can they be optimised?**

# Hardware Locks

- **Atomic Instructions**
  - > **Compare And Swap**
  - > **Lock instructions on x86**
    - **LOCK XADD is a bit special**
- **Used to update *sequences* and *pointers***
- **What are the costs of these operations?**
- **Guess how software locks are created?**

# **Let's Look At A Lock-Free Algorithm**

# Single Producer – Single Consumer Queue

```
public final class ConcurrentArrayQueue<E>
    implements Queue<E>
{
    private final E[] ringBuffer;

    private volatile int addedCounter = 0;
    private volatile int removedCounter = 0;

    public ConcurrentArrayQueue(final int size)
    {
        ringBuffer = (E[])new Object[size];
    }
}
```

# Single Producer – Single Consumer Queue

```
public boolean offer(final E e)
{
    if (addedCounter - removedCounter ==
        ringBuffer.length)
    {
        return false;
    }

    ringBuffer[addedCounter % ringBuffer.length] = e;
    addedCounter++;

    return true;
}
```

# Single Producer – Single Consumer Queue

```
public E poll()
{
    if (addedCounter == removedCounter)
    {
        return null;
    }

    int removeIndex =
        removedCounter % ringBuffer.length;

    E element = ringBuffer[removeIndex];
    ringBuffer[removeIndex] = null;
    removedCounter++;

    return element;
}
```



**Let's Apply Some**  
***“Mechanical Sympathy”***

## **Mechanical Sympathy In Action**

- **Power of 2 Queue Size**
- **Padded counters to prevent false sharing**
- **Avoiding lock instructions on volatile operations**

## Single Producer – Single Consumer Queue 2

```
public final class ConcurrentArrayQueue2<E>
    implements Queue<E>
{
    private final int maxSize;
    private final int mask;
    private final E[] ringBuffer;

    private final AtomicInteger addedCounter =
        new PaddedAtomicInteger(0);
    private final AtomicInteger removedCounter =
        new PaddedAtomicInteger(0);

    public ConcurrentArrayQueue2(final int size)
    {
        maxSize = findNextPowerOfTwo(size);
        mask = maxSize - 1;
        ringBuffer = (E[])new Object[maxSize];
    }
}
```

## Single Producer – Single Consumer Queue 2

```
public boolean offer(final E e)
{
    int added = addedCounter.get();
    if (added - removedCounter.get() == maxSize)
    {
        return false;
    }

    ringBuffer[added & mask] = e;
    addedCounter.lazySet(added + 1);

    return true;
}
```

## Single Producer – Single Consumer Queue 2

```
public E poll()
{
    int removed = removedCounter.get();
    if (addedCounter.get() == removed)
    {
        return null;
    }

    int removeIndex = removed & mask;
    E element = ringBuffer[removeIndex];
    ringBuffer[removeIndex] = null;
    removedCounter.lazySet(removed + 1);

    return element;
}
```

# Concurrent Queue Performance Results

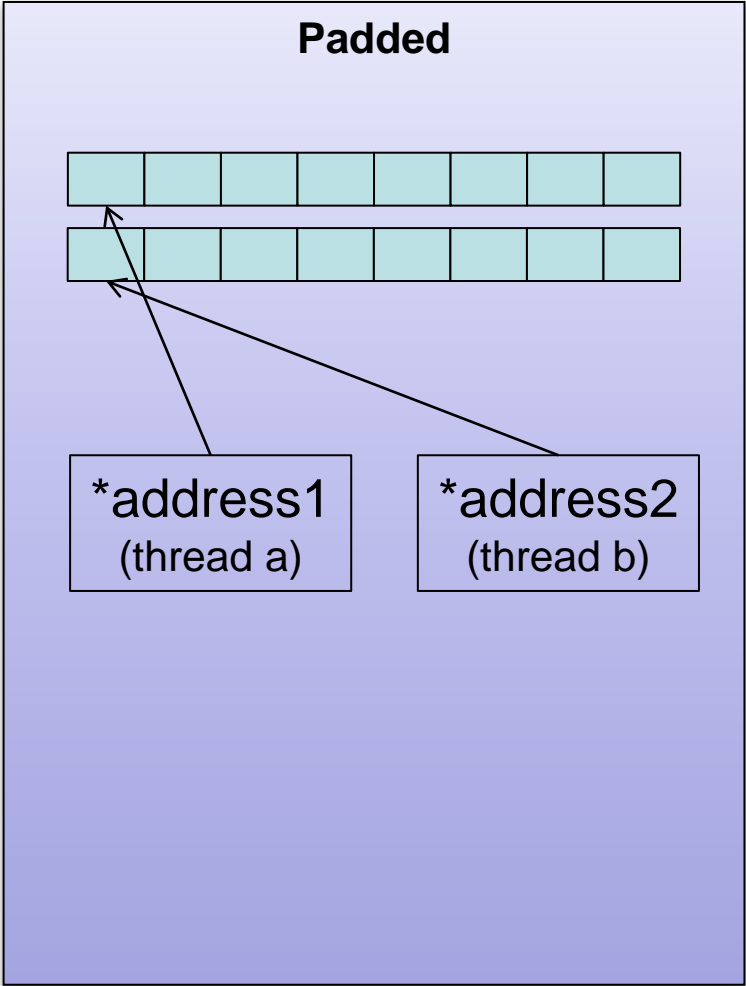
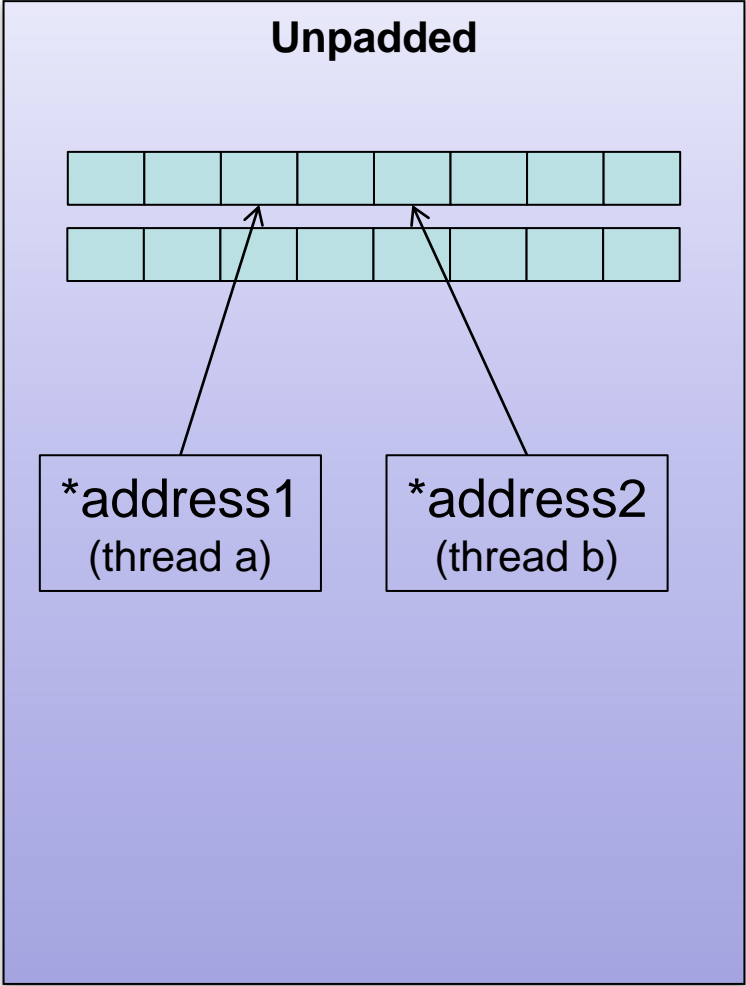
	Ops/Sec (Millions)	Mean Latency (ns)
LinkedBlockingQueue	5	~32,000 / ~500
ArrayBlockingQueue	6	~32,000 / ~600
ConcurrentLinkedQueue	15	NA / ~180
ConcurrentArrayQueue	15	NA / ~120
ConcurrentArrayQueue2	65	NA / ~120

*Note: None of these test are run with thread affinity set*

*Latency: Blocking - put() & take() / Non-Blocking - offer() & poll()*

# False Sharing

# Cache Lines





# False Sharing Test Results

```
int64_t* address = seq->address

for (int i = 0; i < ITERATIONS; i++)
{
    int64_t value = *address;
    value += i;
    *address = value;
    asm volatile("lock addl 0x0, (%rsp)");
}
```

## False Sharing Test Results

	Unpadded	Padded
Million Ops/sec	12.4	104.9
L2 Hit Ratio	1.16%	23.05%
L3 Hit Ratio	2.51%	39.18%
Instructions	4559 M	4508 M
CPU Cycles	63480 M	7551 M
Ins/Cycle Ratio	0.07	0.60

# Signalling

# Signalling

```
// Lock
pthread_mutex_lock(&lock);
sequence = i;
pthread_cond_signal(&condition);
pthread_mutex_unlock(&lock);
```

```
// Soft Barrier
asm volatile("" ::: "memory");
sequence = i;
```

```
// Fence
asm volatile("" ::: "memory");
sequence = i;
asm volatile("lock addl $0x0, (%rsp)");
```

## Signalling Costs

	<b>Lock</b>	<b>Fence</b>	<b>Soft</b>
Million Ops/Sec	9.4	45.7	108.1
L2 Hit Ratio	17.26	28.17	13.32
L3 Hit Ratio	0.78	29.60	27.99
Instructions	12846 M	906 M	801 M
CPU Cycles	28278 M	5808 M	1475 M
Ins/Cycle	0.45	0.16	0.54

# **How Far Can We Go With Lock Free Algorithms?**

## **Further Adventures With Lock-Free Algorithms**

- **State Machines**
- **CAS operations**
- **Wait-Free in addition to Lock-Free algorithms**
- **Thread Affinity**
- **x86 and busy spinning and back off**

## Questions?

Blog (Martin): <http://mechanical-sympathy.blogspot.com/>

Blog (Mike): <http://bad-concurrency.blogspot.com/>

Code: <http://github.com/mikeb01/nonblock>

Twitter: @mjpt777, @mikeb2701

***“The most amazing achievement of the computer software industry is its continuing cancellation of the steady and staggering gains made by the computer hardware industry.”***

***- Henry Peteroski***