Multithreading Deep Dive

a deep investigation on how .NET multithreading primitives map to hardware and Windows Kernel

Gaël Fraiteur

PostSharp Technologies
Founder & Principal Engineer

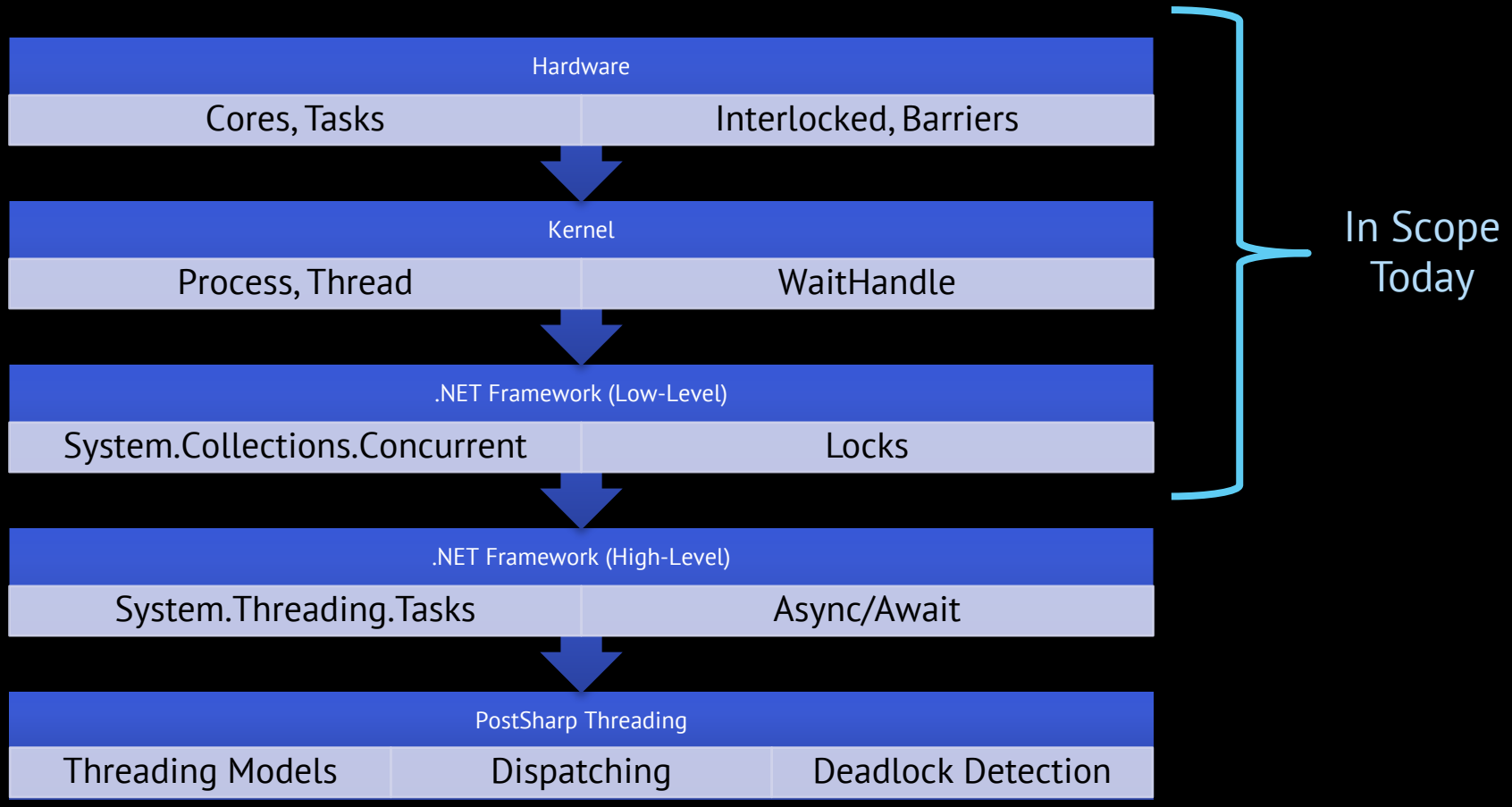
my twitter

@gfraiteur

Objectives

- Deep understanding of multithreading, **from the ground**
- High-performance multithreaded code.

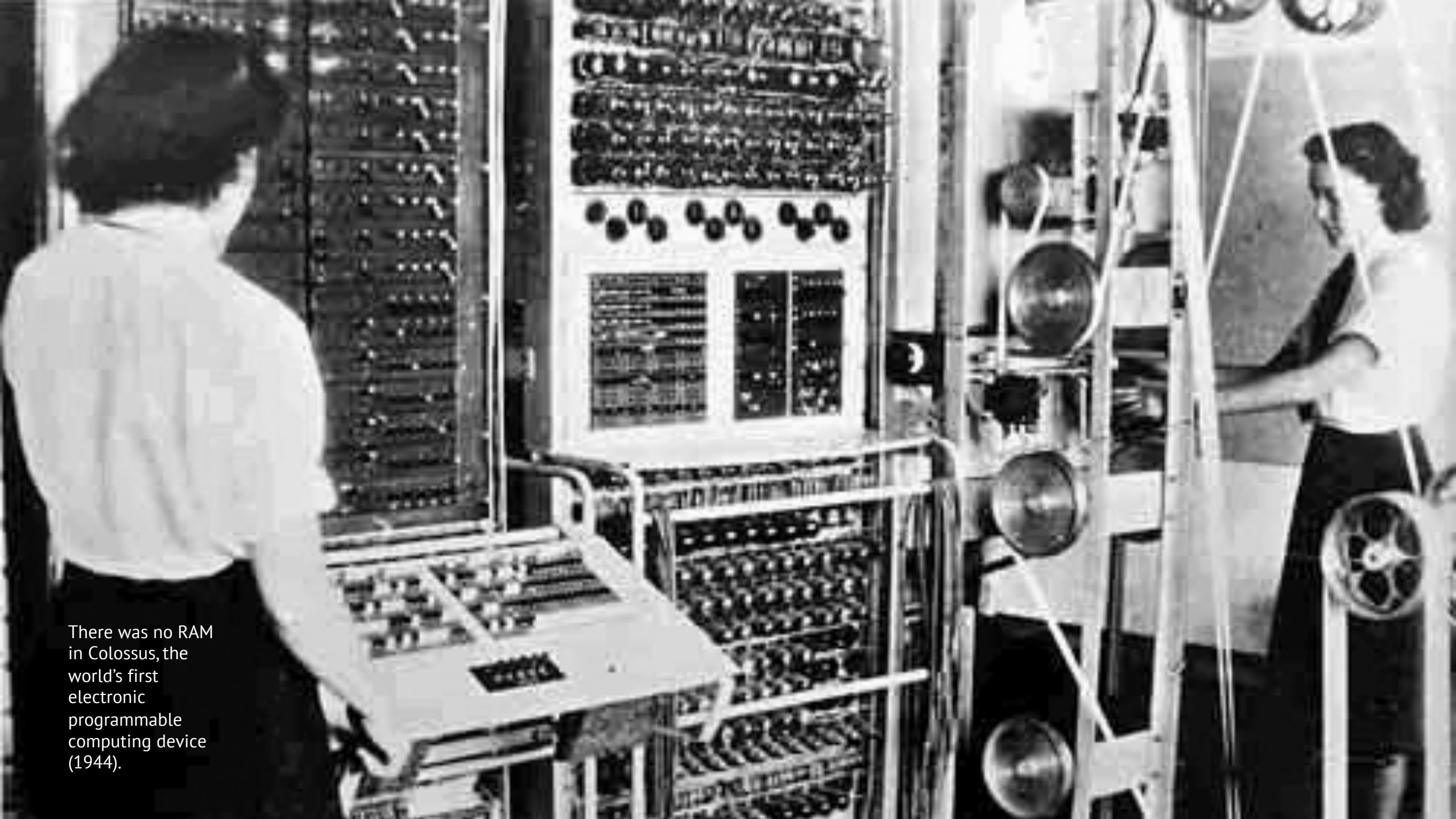
Agenda


Hardware
Cores, Tasks
Interlocked, Barriers
Kernel
Process, Thread
WaitHandle
.NET Framework (Low-Level)
System.Collections.Concurrent
Locks
.NET Framework (High-Level)
System.Threading.Tasks
Async/Await
PostSharp Threading
Threading Models
Dispatching
Deadlock Detection
In Scope Today


Ready?

Hardware

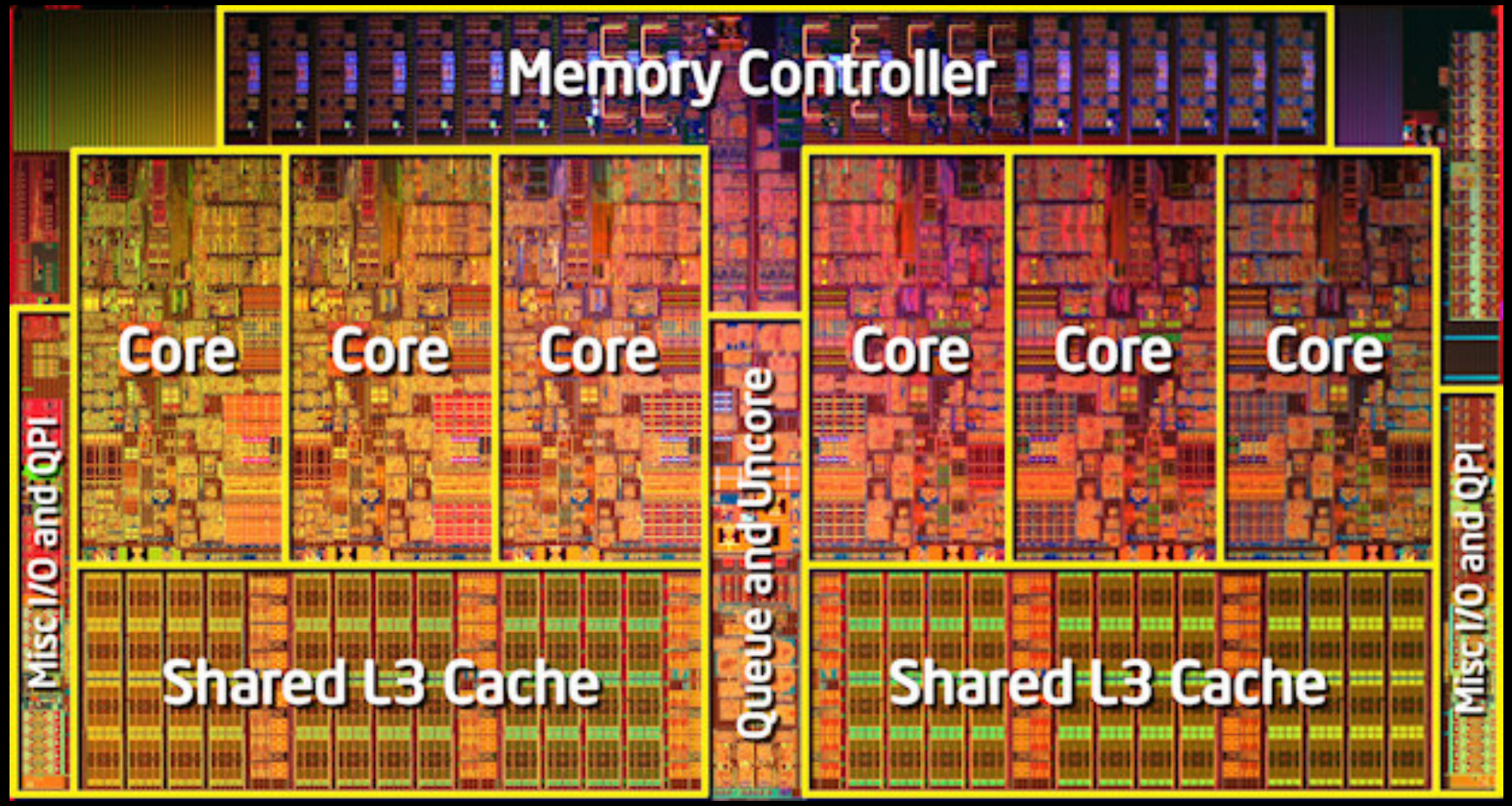
Microarchitecture



There was no RAM in Colossus, the world's first electronic programmable computing device (1944).

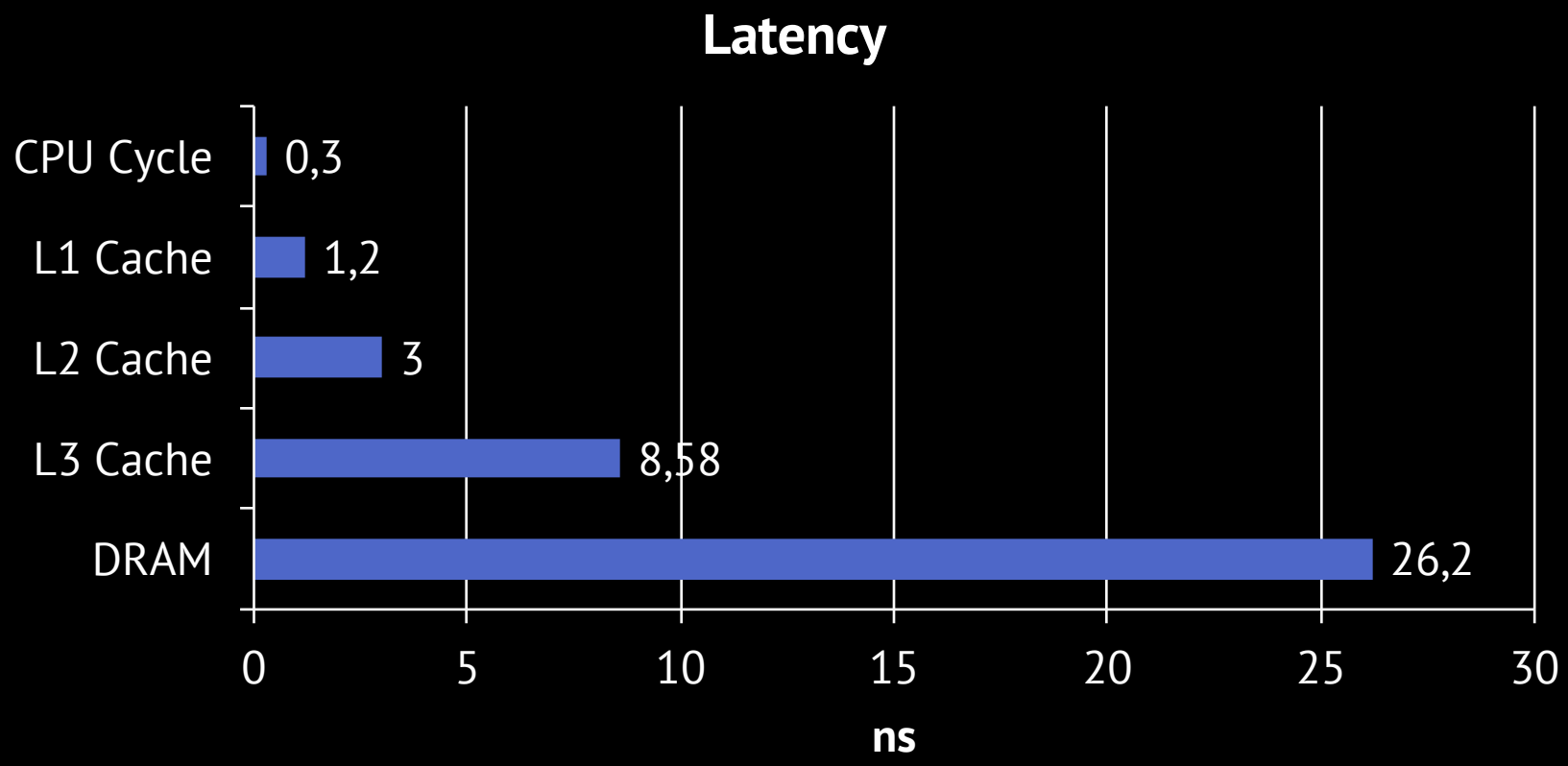
Hardware

Processor microarchitecture



Intel Core i7 980x

Hardware latency



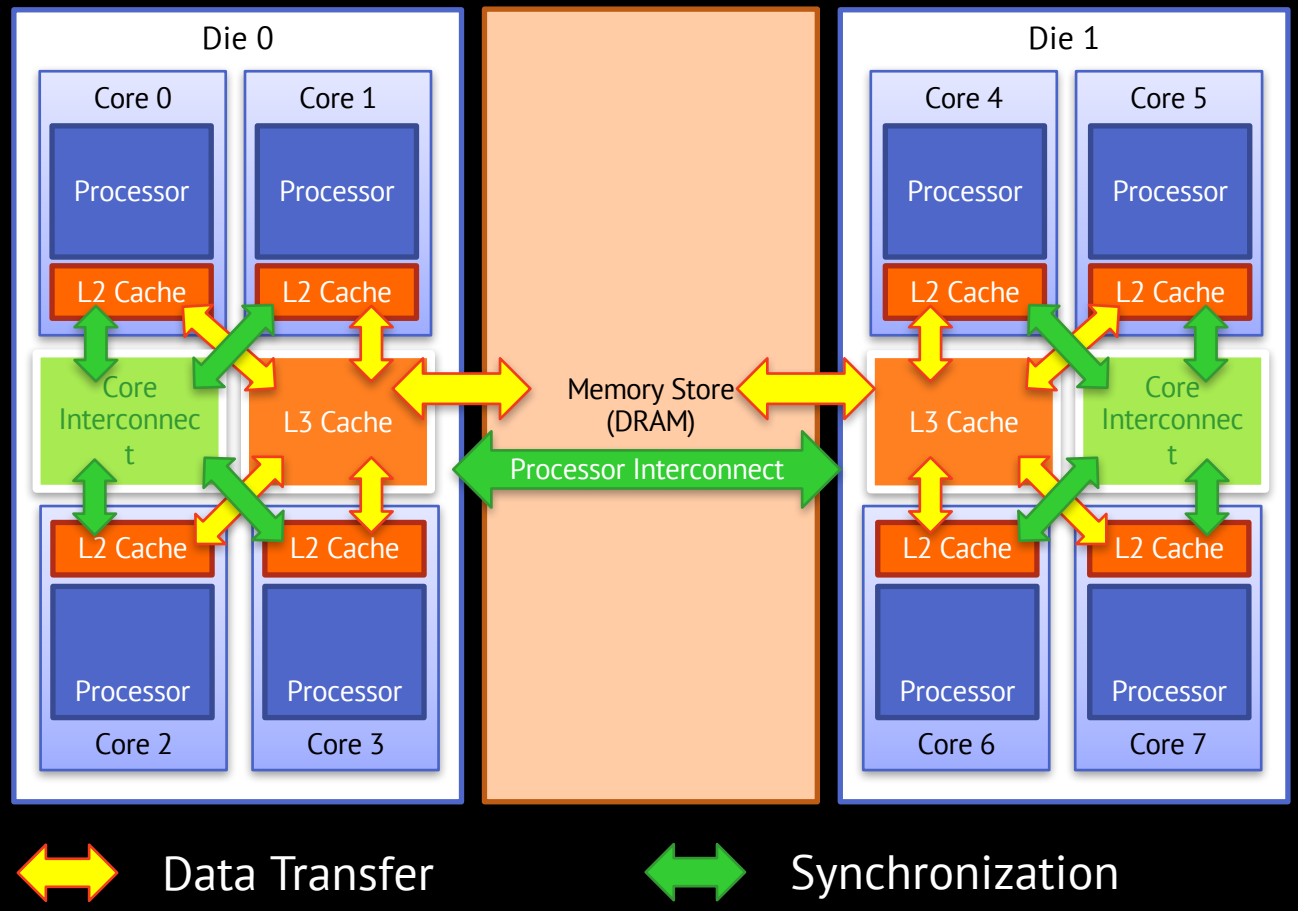
Measured on Intel® Core™ i7-970 Processor (12M Cache, 3.20 GHz, 4.80 GT/s Intel® QPI) with SiSoftware Sandra

Hardware

When Benchmarking…

- Benchmark your production device, not your development notebook
- Plug the power cord

Hardware
Cache Hierarchy



- **Distributed** into caches
  - L1 (per core)
  - L2 (per core)
  - L3 (per die)
- **Synchronized** through an asynchronous "bus":
  - Request/response
  - Message queues
  - Buffers

Hardware

Memory Ordering

- Issues due to implementation details:
  - Asynchronous bus
  - Caching & Buffering
  - Out-of-order execution

Hardware

Memory Ordering

Processor 0	Processor 1
Store A := 1	Load A
Store B := 1	Load B
Possible values of (A, B) for loads?	

Processor 0	Processor 1
Load A	Load B
Store B := 1	Store A := 1
Possible values of (A, B) for loads?	

Processor 0	Processor 1
Store A := 1	Store B := 1
Load B	Load A
Possible values of (A, B) for loads?	

Hardware

Memory Models: Guarantees

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA64	zSeries
Loads reordered after Loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after Stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after Stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after Loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with Loads	Y	Y		Y	Y						Y	
Atomic reordered with Stores	Y	Y		Y	Y	Y					Y	
Dependent Loads reordered	Y											
Incoherent Instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

ARM: weak ordering

X84, AMD64: strong ordering

http://en.wikipedia.org/wiki/Memory_ordering

@gfraiteur

Hardware

Memory Models Compared

Processor 0	Processor 1
Store A := 1	Load A
Store B := 1	Load B
Possible values of (A,B) for loads? • X86: $(A, B) \in \{(0,0), (1,0), (1,1)\}$ • ARM: $(A, B) \in \{(0,0), (1,0), (0,1), (1,1)\}$	

Processor 0	Processor 1
Load A	Load B
Store B := 1	Store A := 1
Possible values of (A, B) for loads? • X86: $(A, B) \in \{(0,0), (1,0), (0,1)\}$ • ARM: $(A, B) \in \{(0,0), (1,0), (0,1), (1,1)\}$	

Hardware

Compiler Memory Ordering

- The compiler (CLR) can:
  - Cache (memory into register),
  - Reorder
  - Coalesce writes
- **volatile** keyword disables compiler optimizations.
  And that's all!

Hardware

Thread.MemoryBarrier()

Processor 0	Processor 1
Store A := 1	Store B := 1
Thread.MemoryBarrier()	Thread.MemoryBarrier()
Load B	Load A
Possible values of (A, B)? • Without MemoryBarrier: $(A, B) \in \{(0,0), (1,0), (0,1), (1,1)\}$ • With MemoryBarrier: $(A, B) \in \{(1,0), (0,1), (1,1)\}$	

Barriers *serialize* access to memory.

Hardware
Atomicity

Processor 0	Processor 1
Store A := (long) 0xFFFFFFFFFFFFFFFF	Load A
Possible values of (A,B)? • X86 (32-bit): $A \in$ {0xFFFFFFFFFFFFFFFF, 0x00000000FFFFFFFF, 0x0000000000000000} • AMD64: $A \in$ {0xFFFFFFFFFFFFFFFF, 0x0000000000000000}	

- Up to native word size (IntPtr)
- Properly aligned fields (attention to struct fields!)

Hardware

Interlocked access

- The only locking mechanism at hardware level.
- System.Threading.Interlocked
  - Add

    ```
    Add(ref x, int a ) { x += a; return x; }
    ```

  - Increment

    ```
    Inc(ref x ) { x += 1; return x; }
    ```

  - Exchange

    ```
    Exc( ref x, int v ) { int t = x; x = a; return t; }
    ```

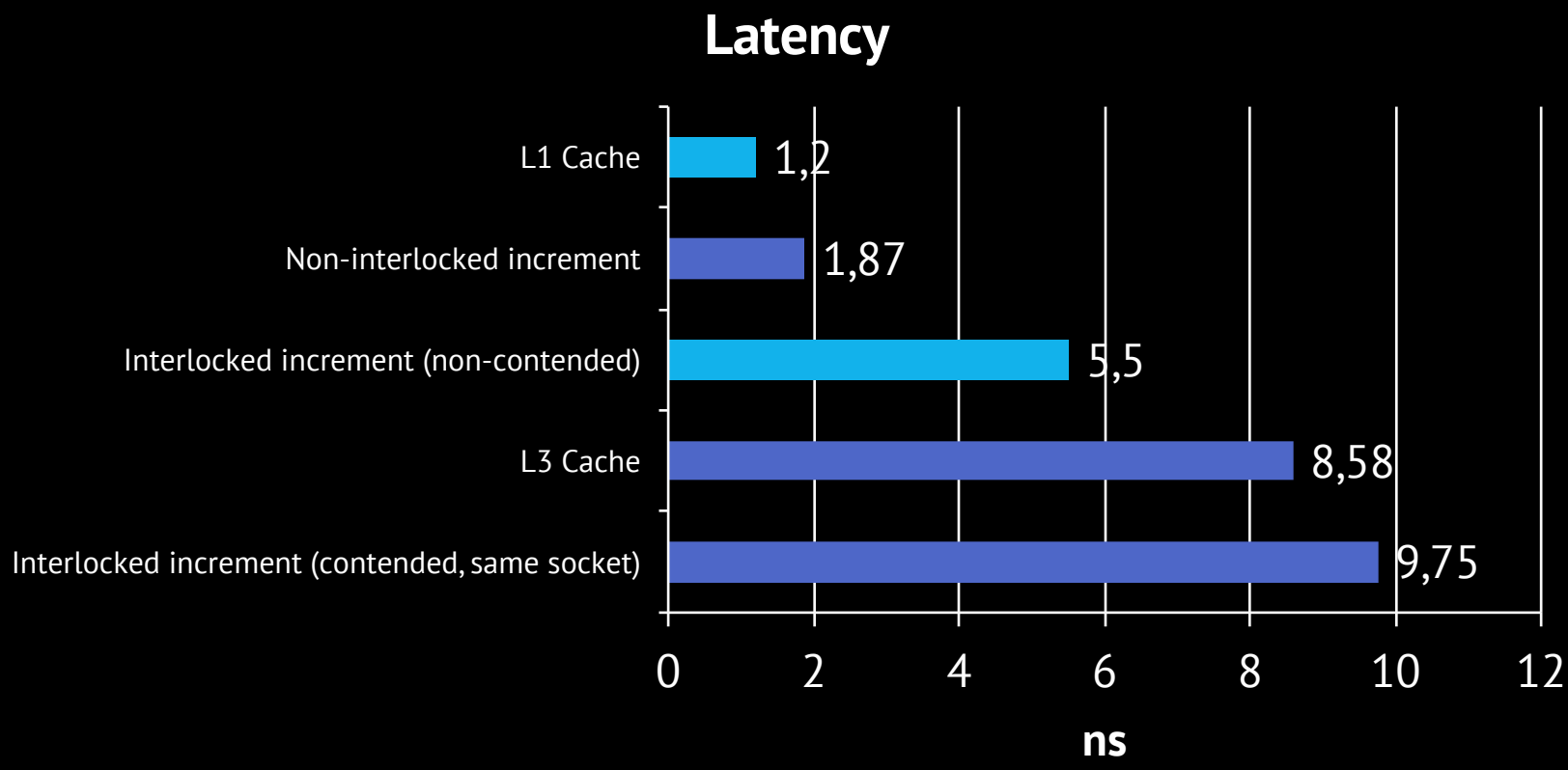
  - CompareExchange

    ```
    CAS( ref x, int v, int c ) { int t = x; if ( t == c ) x = a; return t; }
    ```

  - Read(ref long)
- Implemented by L3 cache and/or interconnect messages.
- Implicit memory barrier

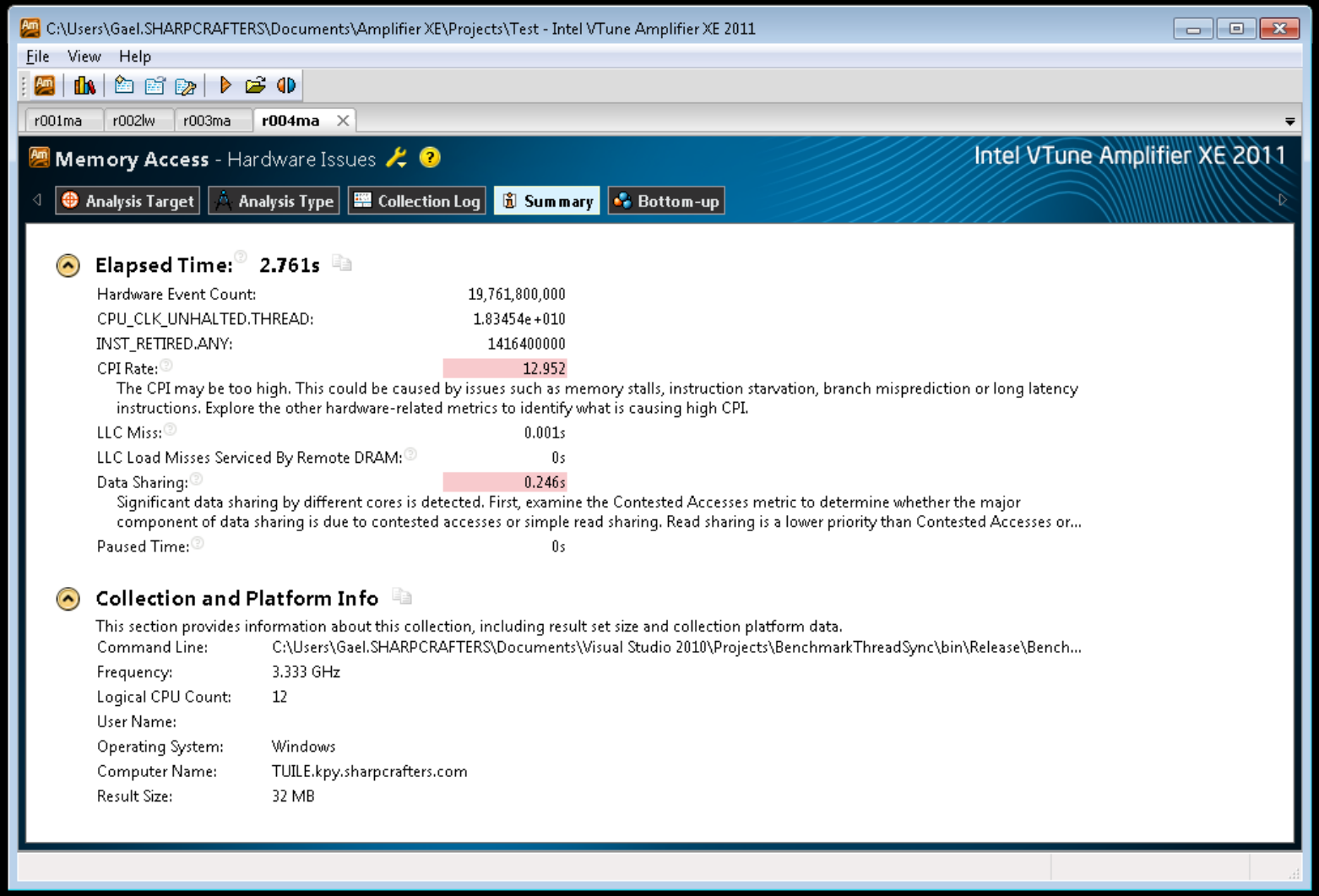
Hardware

Cost of interlocked



Measured on Intel® Core™ i7-970 Processor (12M Cache, 3.20 GHz, 4.80 GT/s Intel® QPI) with C# code. Cache latencies measured with SiSoftware Sandra.

Hardware
Cost of cache coherency



Multitasking

- No thread, no process at hardware level
- There no such thing as “wait”
- One core never does more than 1 “thing” at the same time

  (except HyperThreading)
- Task-State Segment:
  - CPU State
  - Permissions (I/O, IRQ)
  - Memory Mapping
- A task runs until interrupted by hardware or OS

Lab: Non-Blocking Algorithms

- Home-made non-blocking stack

Operating System

Windows Kernel

SideKick (1988). http://en.wikipedia.org/wiki/SideKick

Windows Kernel

Processes and Threads

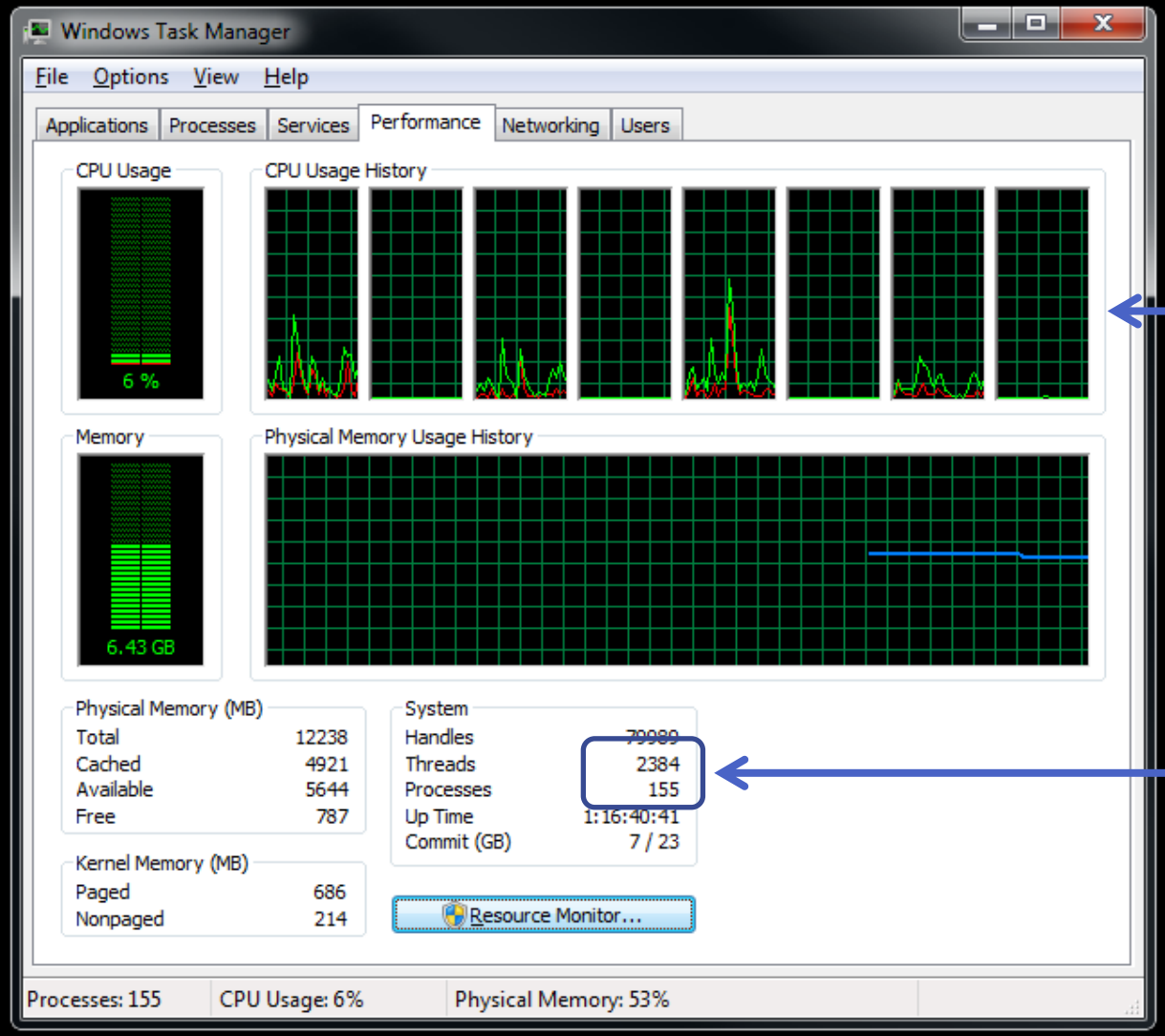
Process

- Virtual Address Space
- Resources
- At least 1 thread
- (other things)

Thread

- Program
  (Sequence of Instructions)
- CPU State
- Wait Dependencies
- (other things)

Windows Kernel
Processed and Threads

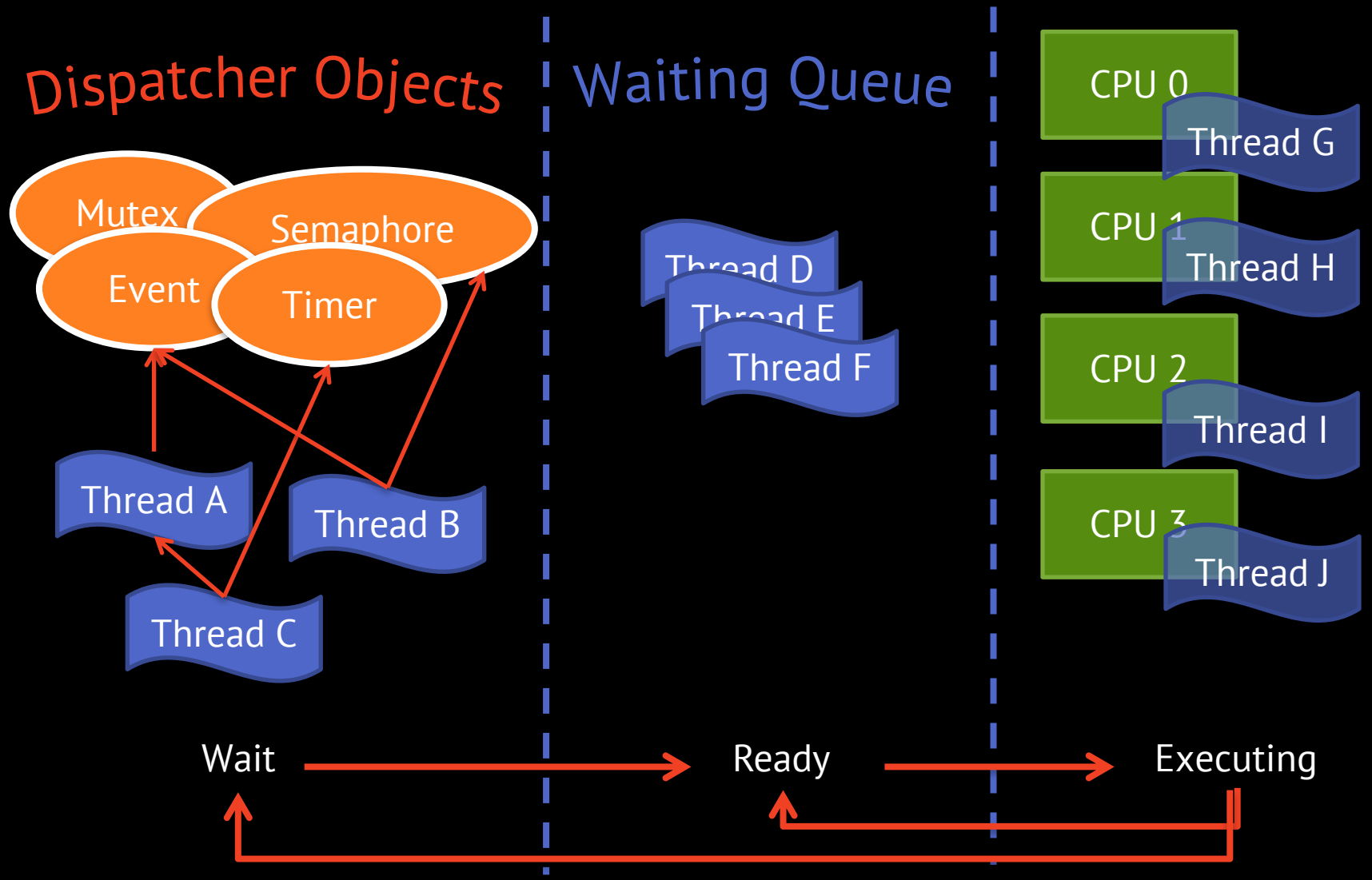


8 Logical Processors
(4 hyper-threaded cores)

2384 threads
155 processes

Windows Kernel

Dispatching threads to CPU



Windows Kernel

Dispatcher Objects (WaitHandle)

Kinds of dispatcher objects

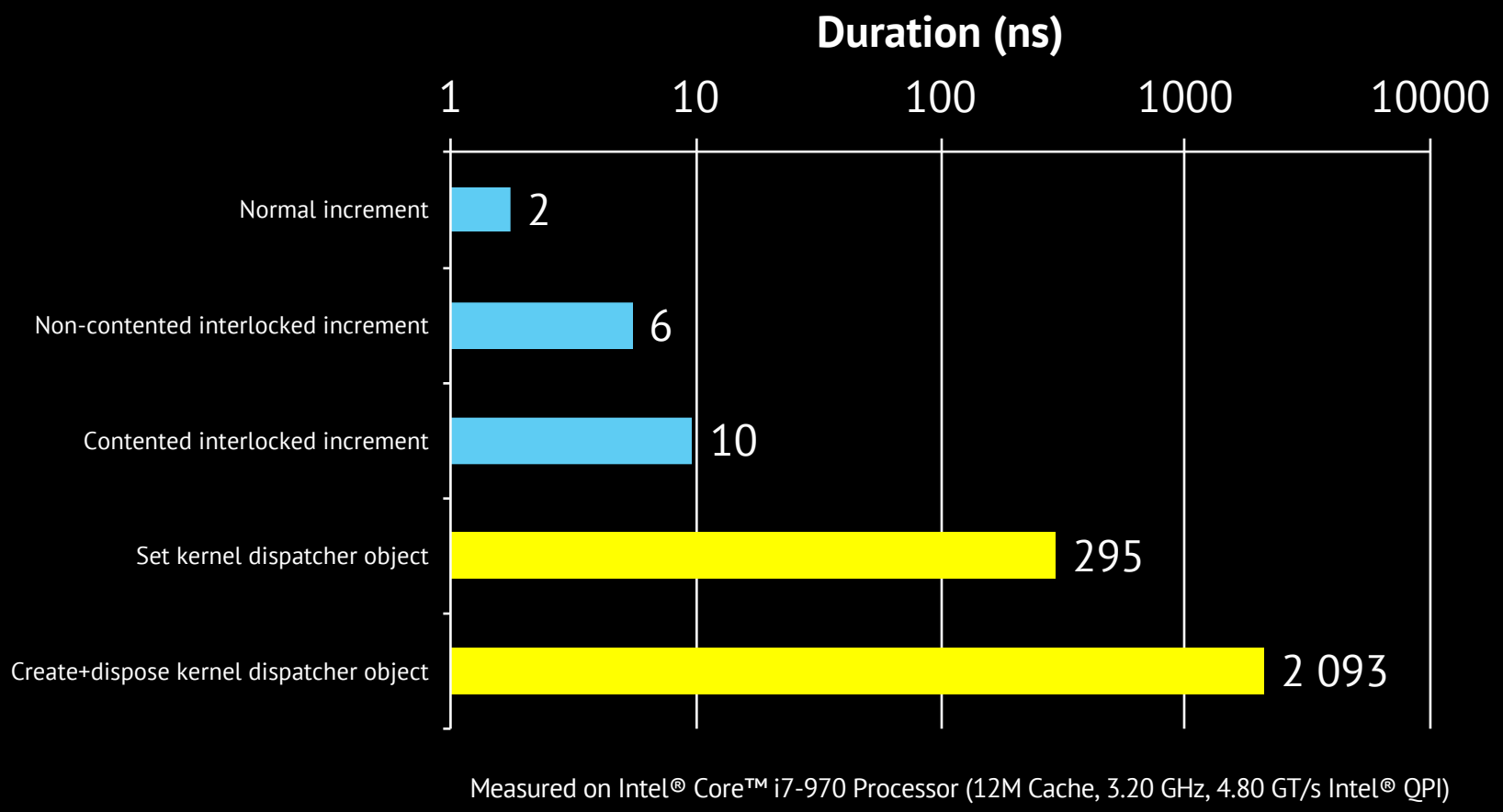
- Event
- Mutex
- Semaphore
- Timer
- Thread

Characteristics

- Live in the kernel
- Sharable among processes
- Expensive!

Windows Kernel

Cost of kernel dispatching



Measured on Intel® Core™ i7-970 Processor (12M Cache, 3.20 GHz, 4.80 GT/s Intel® QPI)

Windows Kernel

Wait Methods

- Thread.Sleep(0) -- give up to threads of same priority
- Thread.Yield -- give up to threads of any priority
- Thread.Sleep(>0) – sleep for 1 ms or more
- Thread.Join – wait for another thread to complete
- WaitHandle.Wait – Wait for a dispatcher object to be signalled
- Thread.SpinWait – Hardware awareness (HT or power management) - *Not a kernel wait method but*

Windows Kernel

SpinWait: smart busy loops

Processor 0	Processor 1

`A = 1;`
`MemoryBarrier;`
`B = 1;`
	`SpinWait w = new SpinWait();`
	`int localB;`
	`if ( A == 1 )`
	`{`
	`  while ( ((localB = B) == 0 )`
	`   w.SpinOnce();`
	`}`

1. Thread.SpinWait (Hardware awareness, e.g. Intel HyperThreading)
2. Thread.Sleep(0) (give up to threads of same priority)
3. Thread.Yield() (give up to threads of any priority)
4. Thread.Sleep(1) (sleeps for 1 ms)

Application Programming

.NET Framework


A piece of cake?

Application Programming

Design Objectives

- Ease of use
- High performance

} **from applications!**

Application Programming

Instruments from the platform

Hardware

- Interlocked
- SpinWait
- Volatile

Operating System

- Thread
- WaitHandle (mutex, event, semaphore, …)
- Timer

Application Programming
New Instruments

Concurrency & Synchronization	• Monitor (Lock)★, SpinLock • ReaderWriterLock★ • Lazy, LazyInitializer • Barrier, CountdownEvent
Data Structures	• BlockingCollection★ • Concurrent(Queue★|Stack★|Bag★|Dictionary★) • ThreadLocal, ThreadStatic
Thread Dispatching	• ThreadPool★ • Dispatcher★, ISynchronizeInvoke • BackgroundWorker
Frameworks	• PLINQ • Tasks • Async/Await

Monitor: the *lock* keyword

1. Start with interlocked operations (no contention)
2. Continue with “spin wait”.
3. Create kernel event and wait

→ Good performance if low contention

Data Structures

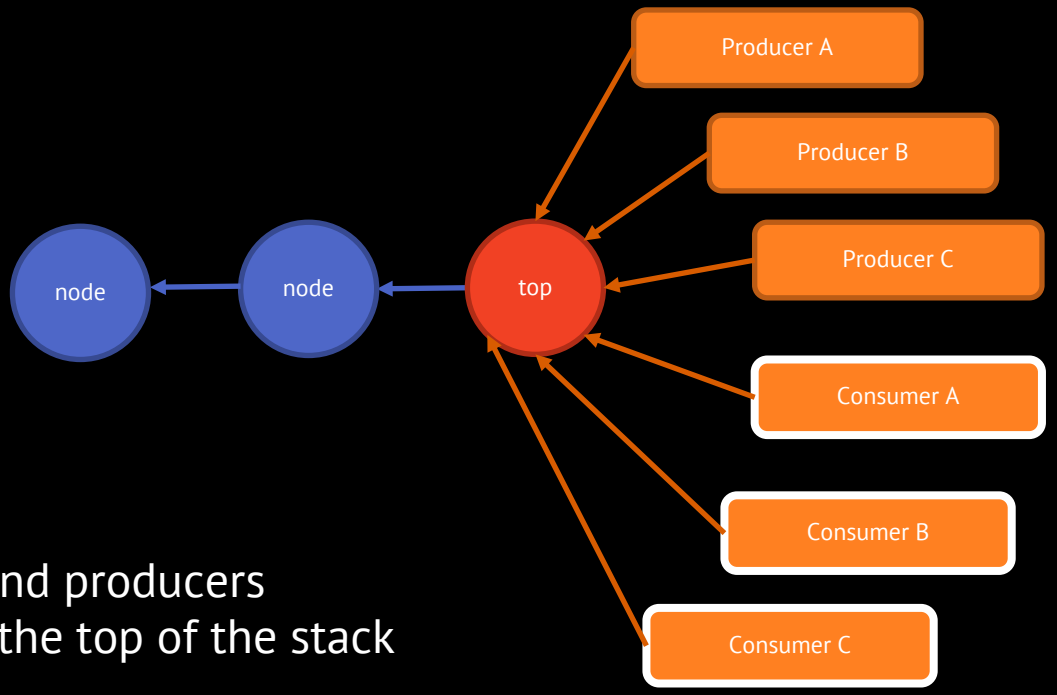
BlockingCollection

- **Use case: pass messages between threads**
- **TryTake**: wait for an item and take it
- **Add**: wait for free capacity (if limited) and add item to queue
- **CompleteAdding**: no more item

Data Structures

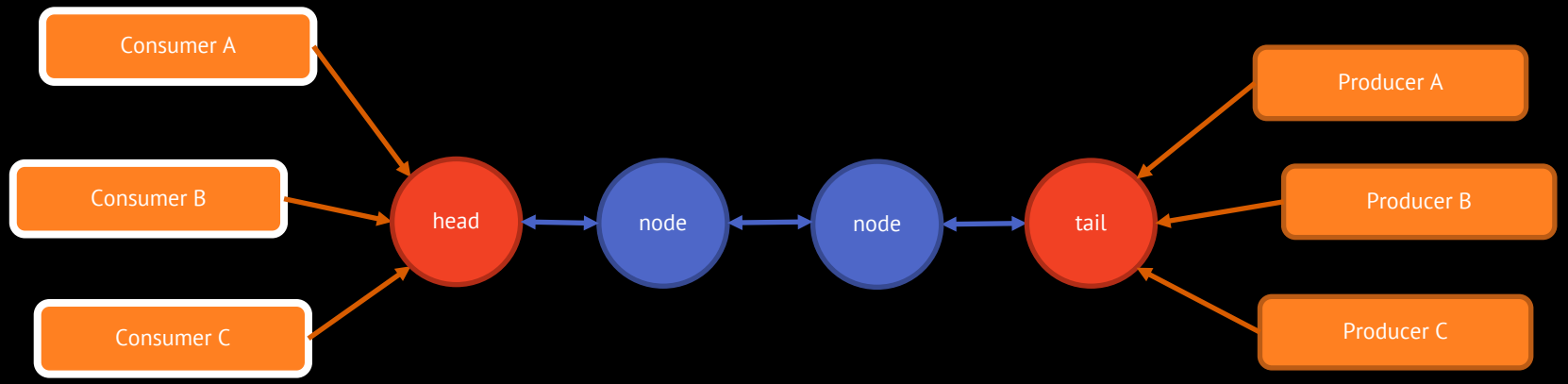
Lab: BlockingCollection

Data Structures
ConcurrentStack



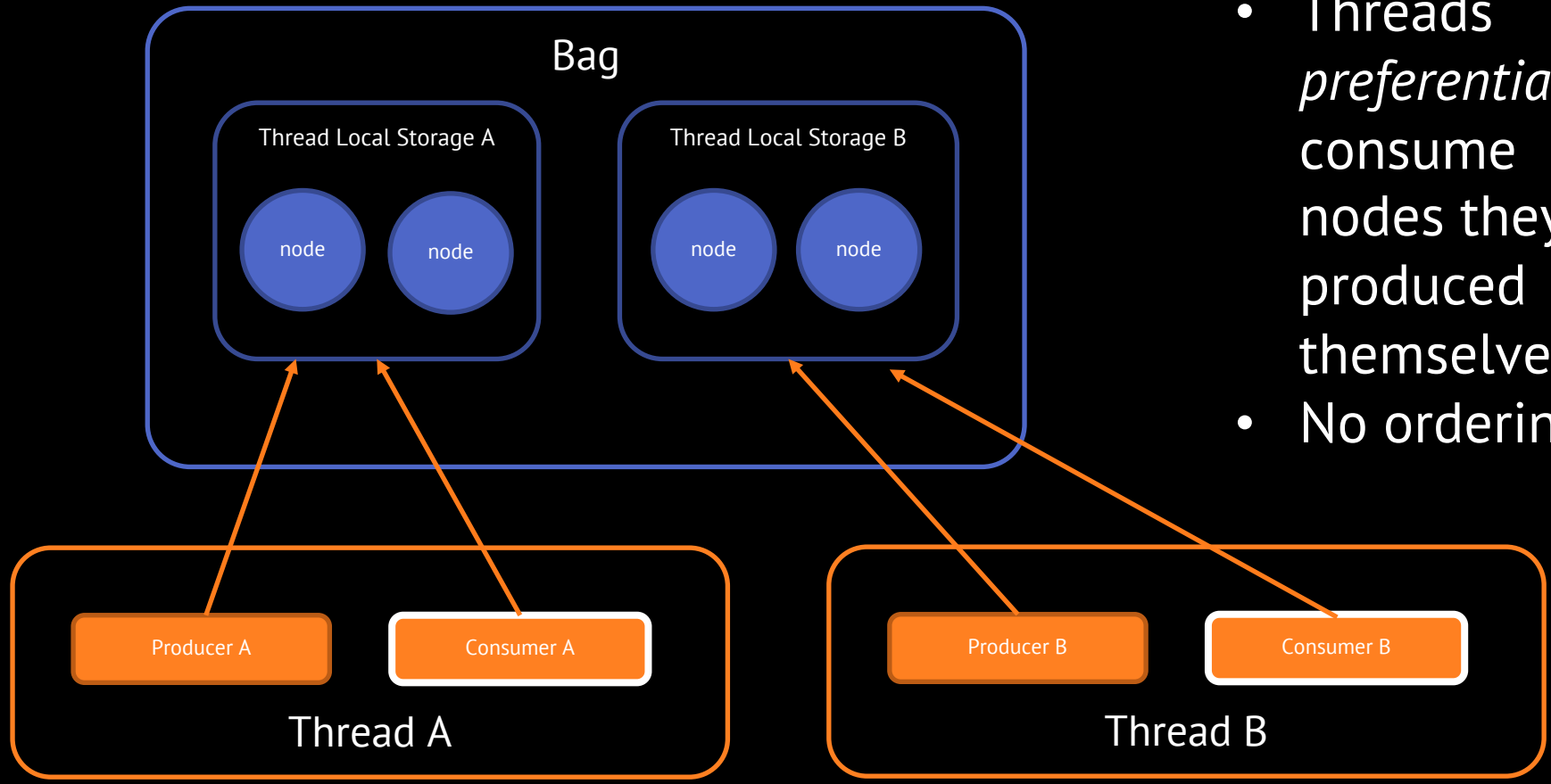
Consumers and producers
compete for the top of the stack

Data Structures
ConcurrentQueue



Consumers compete for the head.
Producers compete for the tail.
Both compete for the same node if the queue is empty.

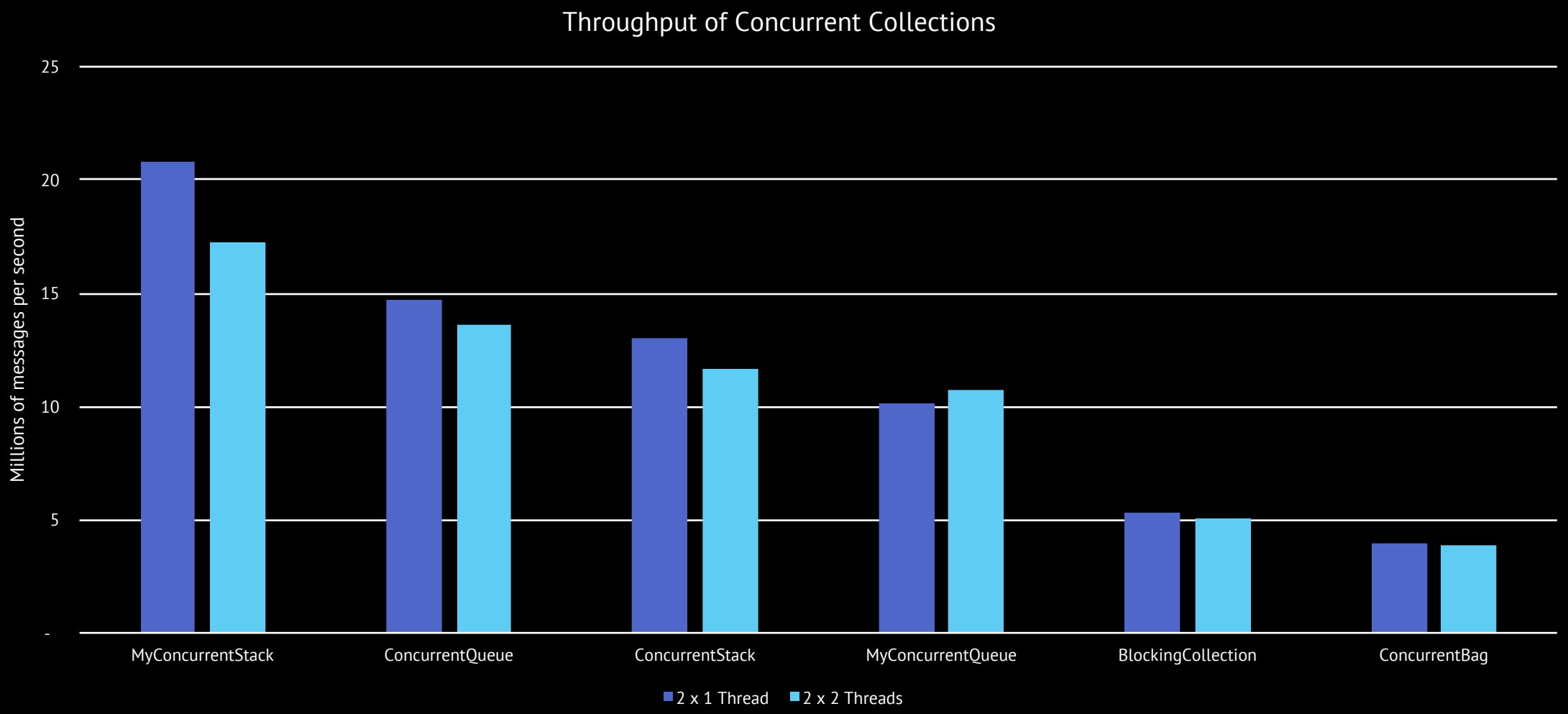
Data Structures
ConcurrentBag



- Threads *preferentially* consume nodes they produced themselves.
- No ordering

Data Structure

Comparing Performance



Data Structures

ConcurrentDictionary

- **TryAdd**
- **TryUpdate (CompareExchange)**
- **TryRemove**
- **AddOrUpdate**
- **GetOrAdd**

Summary

What we've covered today

Covered

- Hardware
- Windows Kernel
- A few .NET Collections

NOT Covered

- .NET Tasks
- .NET async/await
- PostSharp Threading


Thank You!

Errata

- In our implementation of a concurrent stack, the bottleneck is *not* (with current versions of the CLR) memory allocation of nodes and garbage collections.
- A concurrent queue implemented as a linked list is not faster than a concurrent stack because each operation requires two CompareExchange instead of one.
- The system ConcurrentQueue is faster than the system ConcurrentStack because it has a smarter and more complex data structure (linked list of arrays of nodes instead of linked list of nodes).
- Source code: https://github.com/postsharp/ThreadingTalk