Low Latency
C# Programming

Anthony Abate
anthony.abate@gmail.com

Latency

- Latency is the delay from input into a system to desired outcome

- Types:
  - Network / Transmission
  - Disk / Storage
  - Processing

Ground Rules

- C# Programming Domain Only
  - No Kernel Modules / Drivers
  - No Inline Assembly or C/C++
  - No Hardware (GPU, FPGA, ASIC)
  - Specific 'tricks' are done to influence the generated IL / Assembly code
- CPU Bound Problems Only
  - No Network or Disk I/O

Goal: learn techniques you can use without being ‘one with the CPU/hardware’

- Avoiding very ‘low level’ issues
  - CPU Cache problems
  - Memory barriers / fences
  - Instruction re-ordering (hyper threading, optimizer)
  - Memory Paging / Swapping
- references / caveats where appropriate

Faster Hardware Helps

- More Ghz = More CPU cycles per second
- More Cores = More bandwidth

Time scale

Second	Millisecond	Microsecond	Nanosecond	Picosecond
1	10^{-3}	10^{-6}	10^{-9}	10^{-12}

- Low Latency = Micro -> Nano (Fast Code)

- Milliseconds = Slow Code

Overview

- Part I - Measuring Speed
- Part II - Coding (basic)
  - Optimizations / Best Practices
  - Signaling
  - Thread Creation
- Part III - Coding (Advanced)
  - Immutability
  - Advanced Optimization (Caching)
  - Lock-Free Design
  - Producer/Consumer

Part I – Measuring Speed

- Why do we care About Measuring Speed?
  - Test
  - Validate
  - Improve

  - Bottlenecks are not always obvious

How (Not) To Measure (DateTime)

- Very low precision
- Time Scale in milliseconds
- A lot can happen in a Millisecond!

How To Measure (Stopwatch)

- Higher Precision
- Hardware based (Uses High Resolution Timer)
- Timescale is in ticks (not the same as DateTime ticks)
- Scale based on CPU Frequency
  - 1 Mhz = 1 tick
  - 1 Ghz = 1000 ticks

```
double ticks = sw.ElapsedTicks;
double seconds = ticks / Stopwatch.Frequency
double milliseconds = (ticks / Stopwatch.Frequency) * 1000
double nanoseconds = (ticks / Stopwatch.Frequency) * 1000000000
```

Technique 1 (Tight Loop)

- Single-Threaded
- Averages time
- Can infer sub-tick level timings
- Does not measure directly
- Does not measure “Latency”
- Not “realistic”
- Between Start / Stop is a Mystery

```
var sw = new Stopwatch();

sw.Start();

for (int i = 0; i < 10000000; ++i)
{
    // Do Something
}

sw.Stop();
```

Datetime vs Stopwatch Resolution

- Code Demo

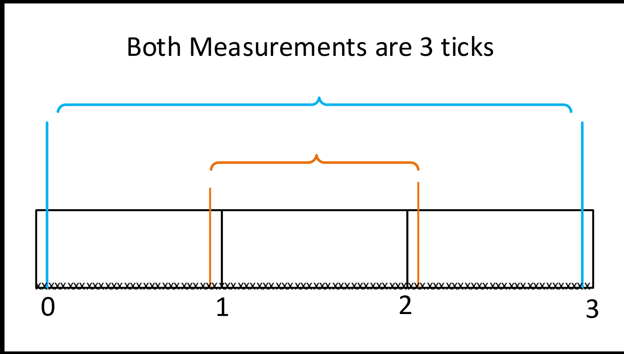
Technique 2 (Latency)

- Multithreaded
- Close to “Actual time”
- 1 Tick level Resolution
- Directly measures code blocks
- measures “Latency”
- “Real world” Scenarios

```
var sw = new Stopwatch();

//Thread 1
sw.Start();

//--- (Signal) -------

//Thread 2
sw.Stop();
```

1 Tick = Latency Resolution

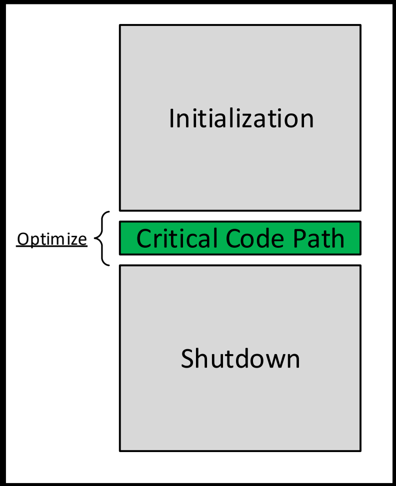
- 1 Tick = ~300 nanoseconds (my computer)
- A lot happens in a tick
- .. But.. Its “good enough”
- If you have latency, it will be MUCH HIGHER than 300ns
- Bottle necks become obvious



Part II - Coding (basic)

- “Common Sense”

- Program Code can be split into 3 Categories:
  - Startup / Initialization
  - Critical Path
  - Shutdown / Post Initialization



- Only optimize the “critical path”

- Correct Data structure / Algorithm choice for problem

Pre-Allocate

- Bad to use ‘new’ in the critical code path
  - Takes ‘time’
  - can trigger possible Garbage Collection
- Requires Heap
  - Shared among all threads => Requires synchronization

Pre-Initialize (Eager Loading)

- Some objects have long startups
- Pre Start threads
- Pre Open connections
- Factory / Builders Patterns

Use Integer Types

- Fits into normal CPU registers
  - Everyone is 64 bit now
- Interlocked / Atomic operations
- Assembly opcode optimized
  - Arithmetic
  - Bit Operations

- Bool (1 bit)
- Byte / Sbyte (8bit)
- Ushort/ Short (16bit)
- Uint / Int (32bit)
- Ulong Long (64bit)

Enums are integers Too!

- Enums are value types (stack allocated)
- Can be used to assign meaning to numbers
- Flags
  - Can be used for bitmask
  - BIT Operations

Integer Type safety

- Type safety will help you find bugs via the compiler
- Use unsigned variants
  - For loop counters
  - Array indexers
  - Age fields
- Enums
  - Only allow the defined values
- Bool
  - True/false (1 bit)

Avoid Floating Point

- Requires special CPU registers
- Possibly Larger than native CPU Register
- Operations require more CPU Cycles

- Float (32bit)
- Double (64bit)
- Decimal (128bit)

Don’t use strings!

- String operations are very slow
- Arbitrary size means unknown performance
- Heap allocated
- CPU operations not optimized
- Immutable
  - intermediate strings may be created with every operation

Seriously… Don’t Use Strings

- Use the ‘initialize’ phase of your code to convert strings to integers
- Use the ‘shutdown’ phase to build strings / messages

Minimize String Usage

- OK… ok…

- if you really 'need' it…

Use The Stack…

- Stack variables avoid the heap
- Value Types (Structs, Enums)
- Stackalloc (similar to ALLOCA() in C++)
- Thread Local Storage (TLS)

Use Arrays

- Indexed using integers
- Can be stack allocated
- The basis of many non-blocking / lock free data structures
- Can be ‘thread-safe’ without locking

Shutdown / Time after Critical path

- While your program is ending, or coming out of time critical path use that time to extract more information
- Convert timestamps to DateTimes
- Convert Integers to strings for logging
- Do string operations if needed

Logging Dilemma

- Logging is good for diagnosing Production code
- Real-time / real-world timing data needed
- Logging is usually very very slow (string + IO operations)

“Observer Effect”

- “changes that the act of observation will make on a phenomenon being observed”

- Tools Profiling Code
- Excessive Logging

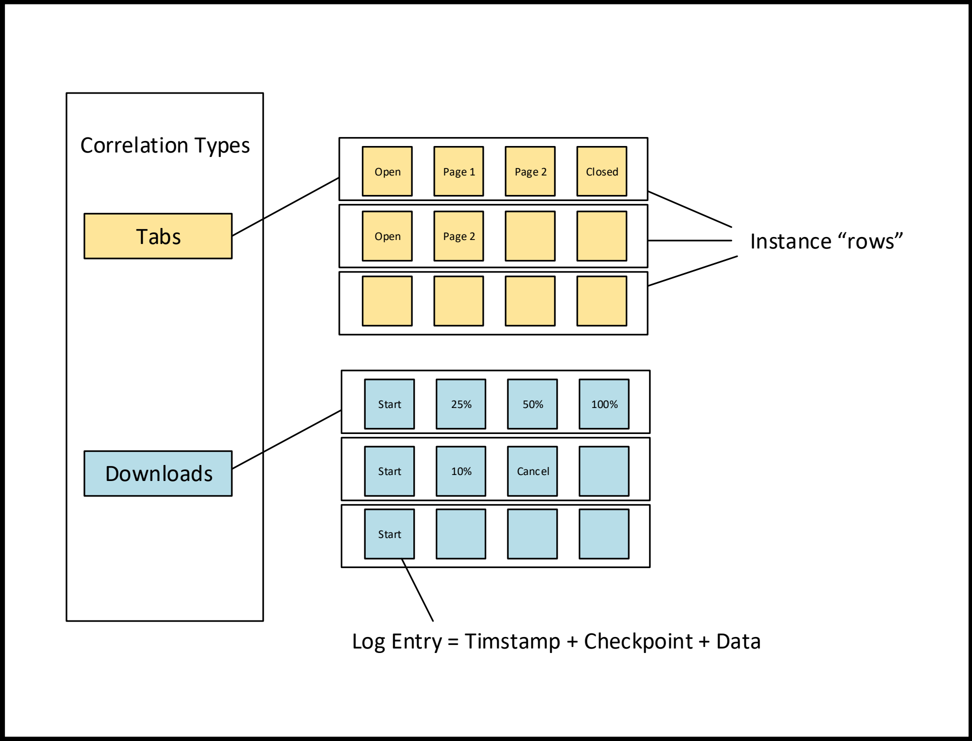
Log4net Overhead

- Code Demo

- I use log4net minimally
  - Error logging
  - Interesting ‘events’

Nano Logger (low Latency)

- Can build a simple logger using all the previous techniques combined
- Record
  - Timestamp (ticks)
  - Code Location (predetermined)
- 4 level Tree
  - Correlation Type
  - Instance Id
  - Checkpoint
  - Checkpoint Detail



Correlation Types
Tabs
Downloads
Open
Page 1
Page 2
Closed
Open
Page 2
Instance “rows”
Start
25%
50%
100%
Start
10%
Cancel
Start
Log Entry = Timstamp + Checkpoint + Data

Nano Logger

- Pre-Init / Pre-Allocate
  - Arrays of Arrays
- Integers only
- No allocations
- Non-Blocking, No Locks
- Shutdown for analysis

Nano Logger performance

- NanoLogger vs Log4net
- 16-20ns vs 3-5ms (300,000x faster)
- 300 ns vs 3ms (10,000x more accurate)
- **5-6 Orders of Magnitude Difference!**

Notes About Testing

- First Execution of code may be very slow
  - Loading Assemblies
  - JIT
  - Static Initialization
- Debug vs Release
- IL Optimization
  - May re-order your code!

.Net Performance (Demos)

- Increment
- Tasks / Threads
- Signal / Notify threads

.Net Performance Conclusion

- Increment:
  - Fastest is no lock (hard to design… but we'll revisit this soon)
  - Interlocked fastest 'thread safe' when no competition
  - Lock => surprisingly, slightly <u>faster</u> than interlocked for multiple threads!
- Task / Create
  - Best to have dedicated threads already running / pre-initialized
- Signaling
  - Busy/while loop is fastest, but dedicates a thread to a core

Part III - Coding (ADVANCED CONCEPTS)

Cores / Threads / Context switches

- Manually create and manage threads
- Dedicate cores to specific work threads
- Set Thread priority where appropriate
- Be mindful that context switches can occur if system is overloaded

Context Switch

- Hard to avoid them without manually balancing thread/core workloads
- Can’t predict them
- Can’t measure them… directly

- Code Demo

Unroll functions / Loops (maybe)

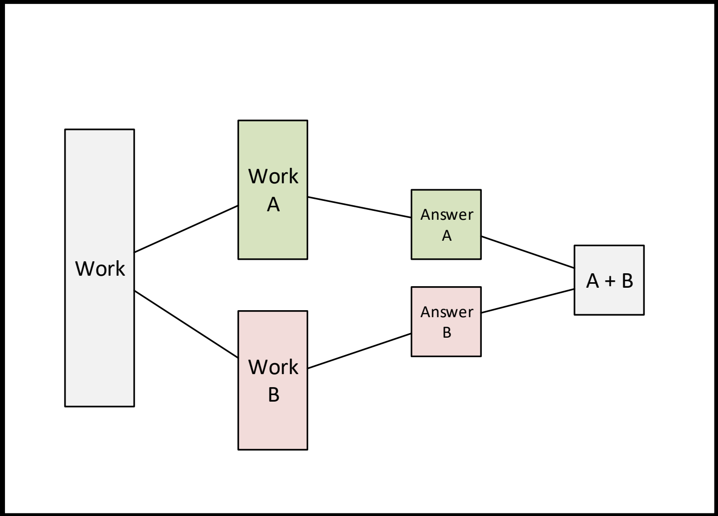
- C# doesn’t have “inline” like C++
- JITer will inline functions
- “Aggressive Inline”
- Unroll for loop / counter
- Static vs Non-Static function calls

Immutable Designs

- Single assignment
- Value can never change
- Avoid blocking / concurrency issues
- Erlang (RabbitMQ)

Partition & Merge (Divide & Conquer)

- Split work into “immutable” partitions
- Partitions should be isolated and Non-overlapping
- All work to complete should be present
- Differ merge till after critical code path
  - (threads can log locally, on shutdown merge with parent)
- Merge surface area should be as small as possible
  - Record Version #



DB Transaction (optimistic)

- Databases use optimistic concurrency
- Unlikely 2 clients update same data at once
- No need for ‘heavy lock’
- Rows have a ‘version number’
  - SQL “Timestamp” type
  - very small merge ‘surface area’

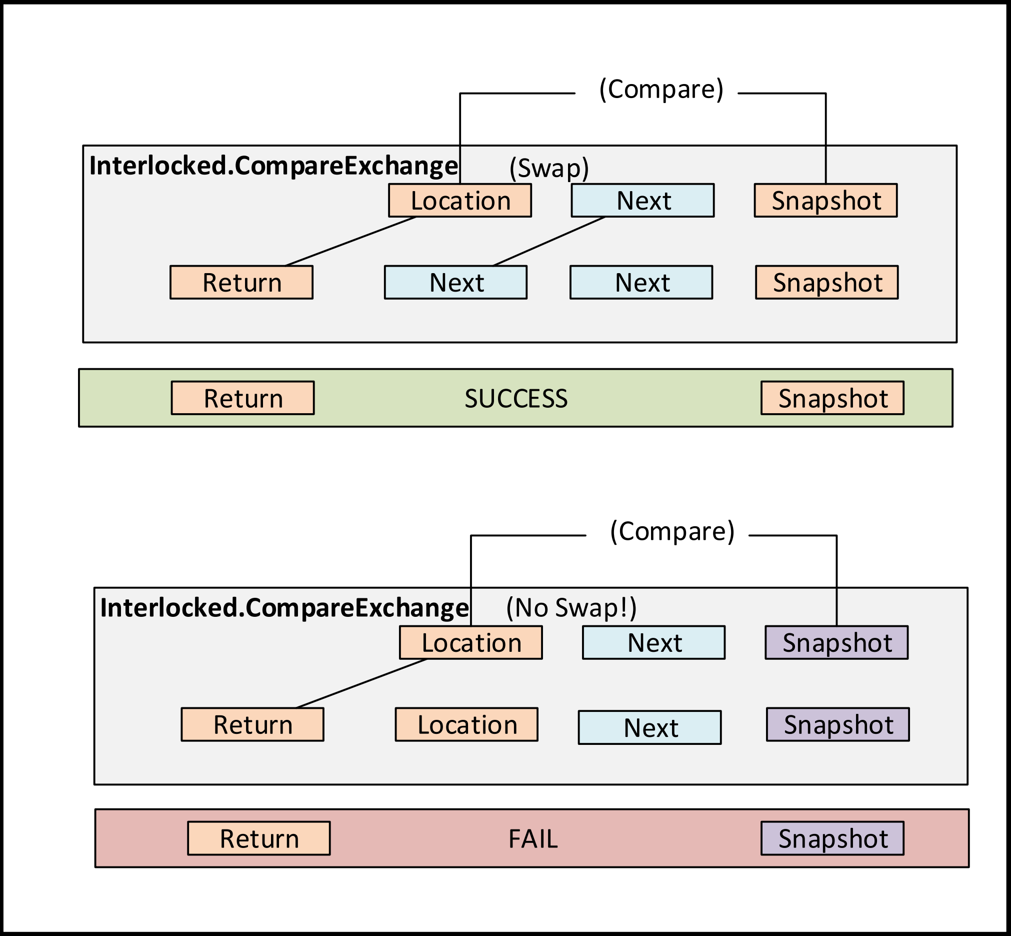
- 1. Client caches data with version
- 2. Client does work on data
- 3. Client submits data back with same version
- 4. if DB version == Client Version => Commit
  - else Concurrency Error!
- 5. Client must start over (Store Wins)

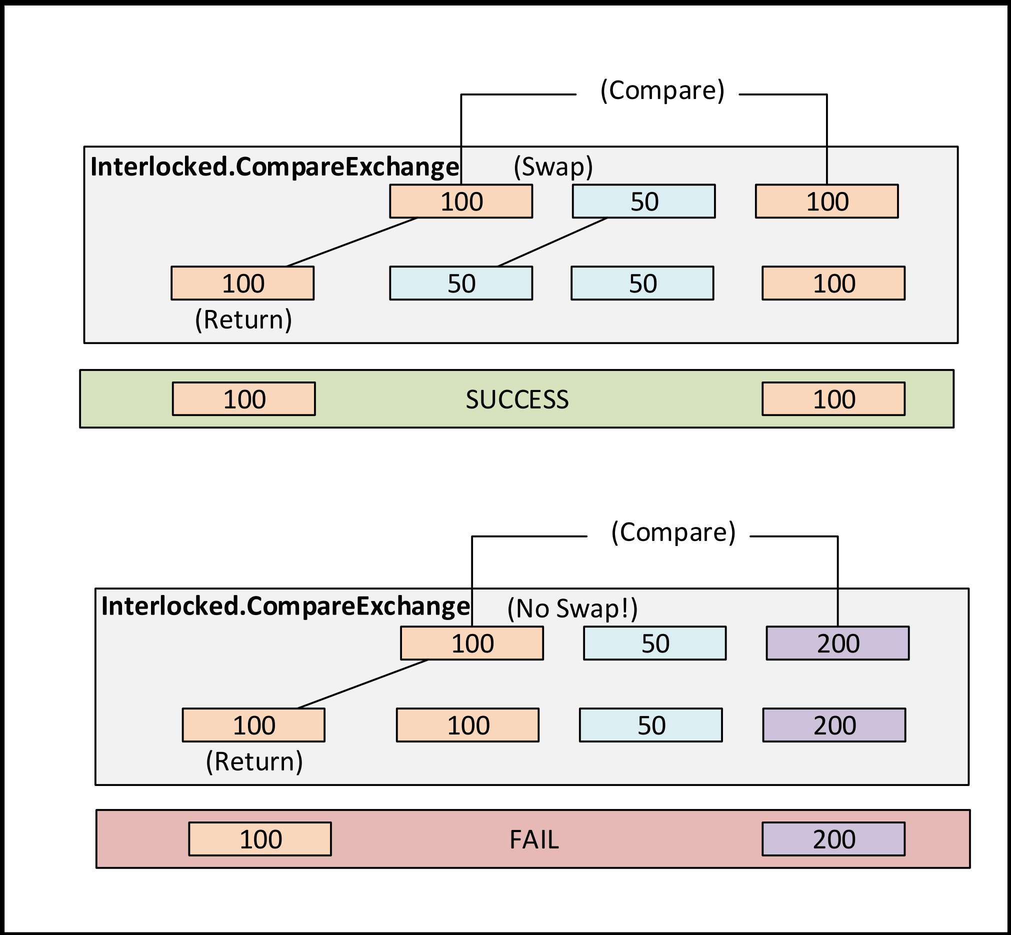
```
while(true)
{
    var versionSnapshot = this.storage.Snapshot();

    var next = this.OnComputeNext(versionSnapshot, data);

    // commit checks version numbers
    var committed = this.storage.Commit(next, versionSnapshot);

    if (committed)
    {
        return true;
    }
}
```

CompareExchange

- Very confusing…
- but behaves just like
- DB Transaction

```
public static int CompareExchange(
        ref int location1,
        int value,
        int comparand
```




(Compare)
Interlocked.CompareExchange
(Swap)
100
50
100
100
50
50
100
(Return)
100
SUCCESS
100
(Compare)
Interlocked.CompareExchange
(No Swap!)
100
50
200
100
100
50
200
(Return)
100
FAIL
200

Caching / Memoization

- Store values that require computation for future use
- Requires a hash table / dictionary
  - Prefer integer based arrays
- caveats:
  - may require locking
  - If no locking, there might be duplicate initialization
  - Expiration?
- Memoization – function cache: F(x,y,z) => value

Pre-Caching / Memoization

- Memory is cheap, computers have lots of it
- Have lots of time during ‘start up’
- Precompute all possible outputs based on all ‘expected’ inputs
  - Multi dimensional arrays / jagged arrays
  - cache[var1][var2][var3] => value
  - F(var1,var2,var3) => value (just like memoization)
- Once initialized “read only”
- Thread safe

Lock-free / Non-Blocking Structures

- Combine many of the basic and advance techniques
- Very Hard to write correctly… will demonstrate
- .Net Concurrent collections
  - Danger! Very subtle caveats
  - supplied actions / callbacks can be called more than once!

Newer .NET Collections

Concurrent (Thread safe)

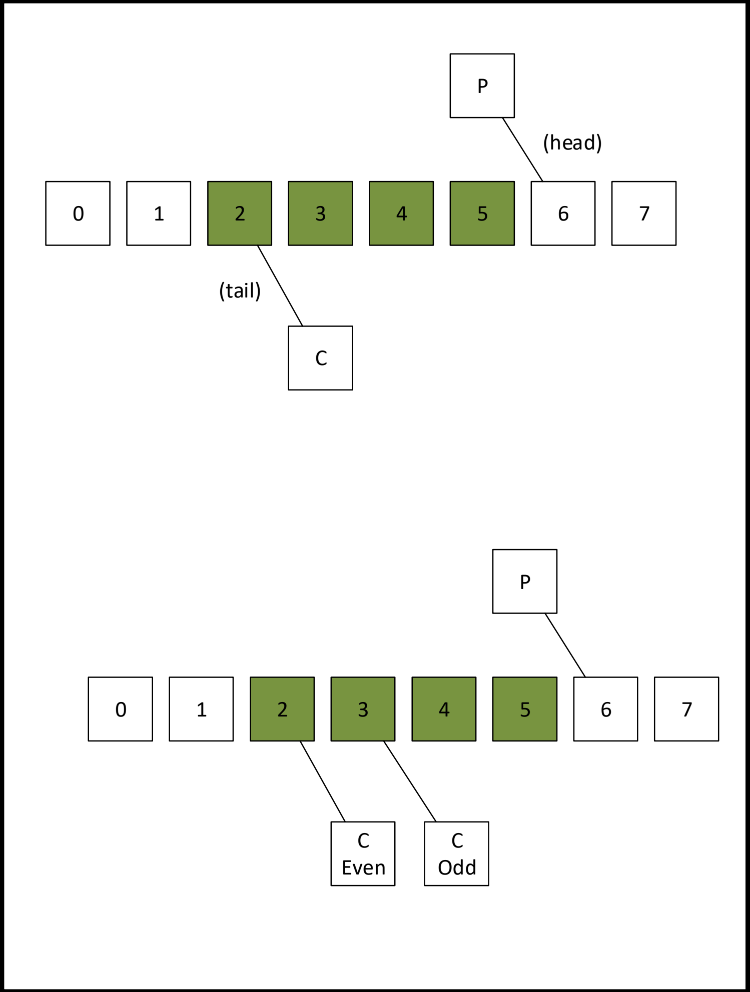
- ConcurrentBag (unordered List)
- ConcurrentDictionary (Hashset also)
- ConcurrentQueue
- ConcurrentStack

Immutable

- ImmutableList
- ImmutableDictionary
- ImmutableQueue
- ImmutableStack
- ImmutableArray
- ImmutableHashSet

Circular Queues

- Queues inherently "thread safe"
  - Read / Write at opposite ends
- 1 produce / 1 consumer (Ring buffer)
- N Producers / M consumers



Advanced Functions / Sync objects

- Spinlock (struct)
- Spinwait (struct)
- yielding