corvalius

# Patterns for High Performance C#

**Federico Lois**
Twitter: @federicolois
Github: redknightlois
Repo: performance-course

**20 billion** operations per day

**20.000.000.000**
operations per day

20.000.000.000 / 24 hours

**833.333.333**

operations per hour

833.333.333 / 60 minutes

**13.888.888**

operations per minute

13.888.888 / 60 seconds

**231.481**

operations per seconds

231.481 / 40 servers

**5.787**

operations per node

231.481 / 1500 servers

**154**

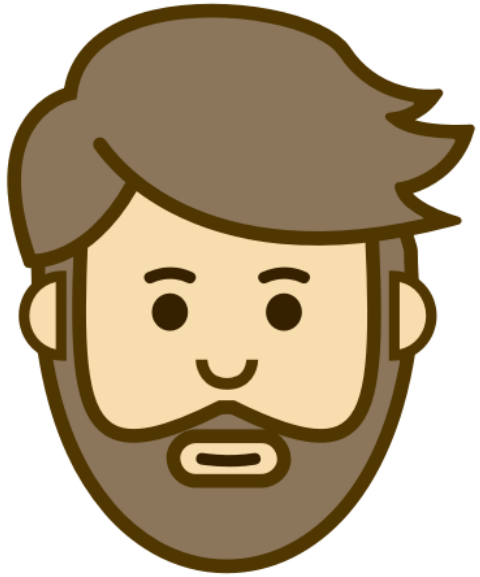
operations per node

# ~ 5 ms

per operation
per second
per node

# When optimizing …

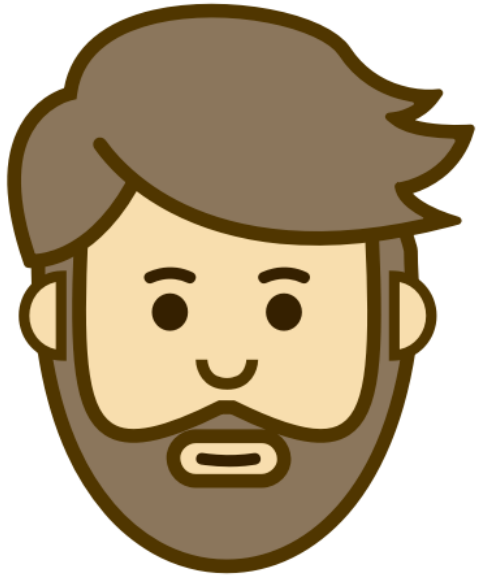
< 600 request/sec per node



~ 6.00**0** request/sec per node

> 60.0**00** request/sec per node

<X*1 metric units per node



~X*1**0** metric units per node

>X*1**00** metric units per node

# Things we all should know (aka the ground truth)

- Interop calls have somewhat constant cost overhead
  - What kills you is high frequency (marshalling)
- Allocations are a major source of performance issues (GC).
- Try-catch and LINQ operators are a no-no
  - Epilog & Prolog cost for exceptions.
  - LINQ method calls & allocations.
  - In high frequency & short operations, you are screwed.
- Accessing a value type via interface will allocate on the heap.
  - Aka Boxing
- Value types are subject to a special set of optimizations.
- Unsafe code is unsafe for a reason.
  - GC pinning is expensive.

# Things we all should know - Inlining

- Compilers directives are only hints.
- Avoiding the call helps you diminish:
  - Instruction cache misses
  - Push/Pop and parameters handling
  - Context switching at the processor level
- Avoiding the call increments
  - Caller size in bytes.
  - Locality of reference
- Collateral effects (among others)
  - Dead code elimination
  - Constant propagation
  - Can be either faster or slower (measure it!!)

# Bottlenecks Rule (aka The Checklist)

## Pareto$^1$

**20%** of the code consumes **80%** of the resources

## Pareto$^2$

**4.0%** of the code consumes **64%** of the resources

## Pareto$^3$

**0.8%** of the code consumes **51%** of the resources

# Bottlenecks Rule (aka The Checklist)

Pareto[1]

**Architecture/Network/Algorithm**

Pareto[2]

**Algorithm Time**

Pareto[3]

**Micro Optimization**

# Bottlenecks Rule (aka The Checklist)

Pareto[1]

**Bottlenecks**

Pareto[2]

**Algorithm Time**

Pareto[3]

**Micro Optimization**

# RavenDB Bottlenecks

Input-Output (2.0)

- Single Write Lock
- Lack of batching ops.

CPU/Memory (2.0)

- Interop cost.
- Hashing.
- JSON processing

# RavenDB Bottlenecks

**Input-Output (2.0)**

- ~~Single Write Lock~~
- ~~Lack of batching ops~~.

**CPU/Memory (2.0)**

- ~~Interop cost.~~
- ~~Hashing.~~
- JSON processing

**Input-Output (3.0)**

- **BTree write-cliff [Bender]**
- **Journal write dominates critical path**
- Single Write Lock

CPU/Memory (3.0)

- JSON processing.
- **Allocations.**
- Virtual calls everywhere.
- **Persistent Data Layout.**
- **Indexing & Thread Scheduling**

On the wild improvement roughly 10x on critical paths

# From 3.x to 4.0

- **Performance First** redesign of the storage engine.
  - Specialized local memory allocators & strings.
  - Zero copy binary serialization format.
  - Managed compression algorithms (LZ4)
    - Avoiding high frequency interop cost.
  - Focus on managing IO/Memory access patterns.
    - Avoid writing to memory/disk if possible.
    - Kernel based prefetching hints.
    - Data layout optimized for locality of references.
  - Removal of most virtual calls.

# RavenDB Bottlenecks

Input-Output (3.0)

- BTree write-cliff [Colton, Farach]
- Journal write dominates critical path
- Single Write Lock

CPU/Memory (3.0)

- JSON processing.
- Allocations.
- Virtual calls everywhere.
- Persistent Data Layout.
- Indexing & Thread Scheduling

# RavenDB Bottlenecks

Input-Output (3.0)

- BTree write-cliff [Colton, Farach]
- Journal write dominates critical path
- Single Write Lock

CPU/Memory (3.0)

- **JSON processing.**
- **Allocations.**
- **Virtual calls everywhere.**
- Persistent Data Layout.
- Indexing & Thread Scheduling

# Assumptions

- We serve requests, a request starts, then it ends.
  - **String allocations dominate by far**
  - Followed [in order] by:
    1. Collections
    2. Gen0 reclaimable assorted objects (fire & forget)
    3. Async machinery
    4. Lambdas captured context.

# Getting rid of allocations

- Unmanaged Strings (circa end of 2015) [P1]
  - Memory Ownership
  - High Frequency allocation
  - Reuse or Renew?
- Pooling by ~~operation~~ request [P1]
  - Great cache/thread locality but bad code locality
- Pooling by ~~context~~ type of object [P2]
  - Great code locality but bad cache/thread locality
- Stack allocation [P3]

# Anatomy of a ByteString

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct ByteString
{
        /// The actual type for the byte string
        public ByteStringType Flags;

        /// The actual length of the byte string
        public int Length;

        /// This is the pointer to the start of the byte stream.
        public byte* Ptr;

        /// This is the total storage size for this byte string.
        /// Length will always be smaller than Size - 1.
        public int Size;
}
```

# SOUNDS FAMILIAR?

```
public struct Span<T>
{
        internal ref T _pointer;
        internal int _length;
}
```

```
public struct Span<byte*>
{
        internal byte* _pointer;
        internal int _length;
}
```

Span-like but legal C# ☺

# Anatomy of a ByteString

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct ByteString
{
        /// The actual type for the byte string
        public ByteStringType Flags;

        /// The actual length of the byte string
        public int Length;

        /// This is the pointer to the start of the byte stream.
        public byte* Ptr;

        /// This is the total storage size for this byte string.
        /// Length will always be smaller than Size - 1.
        public int Size;
}
```

36 Bytes

BUMMER!!!

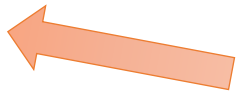
# How would you “fix” this in C/C++?

“There's no problem in Computer Science that can't be solved by adding yet another level of indirection to it”

Andrew Koenig - David Wheeler

# Anatomy of a ByteString

```
[StructLayout(LayoutKind.Sequential)]
public unsafe struct ByteString : IEquatable<ByteString>
{
        internal ByteStringStorage* _pointer;

        internal ByteString(ByteStringStorage* ptr)
        {
            _pointer = ptr;
        }

        public byte* Ptr
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get
            {
                Debug.Assert(HasValue);
                EnsureIsNotBadPointer();

                return _pointer->Ptr;
            }
        }
}
```



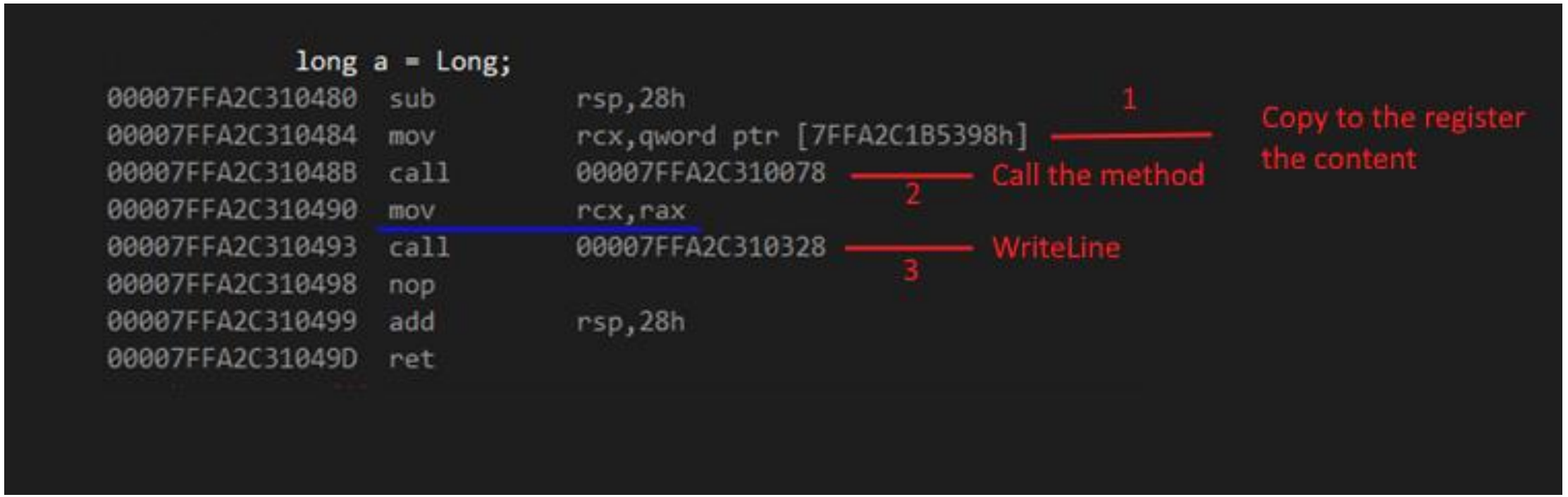
8 Bytes

# How much does passing a ByteString as parameter cost?

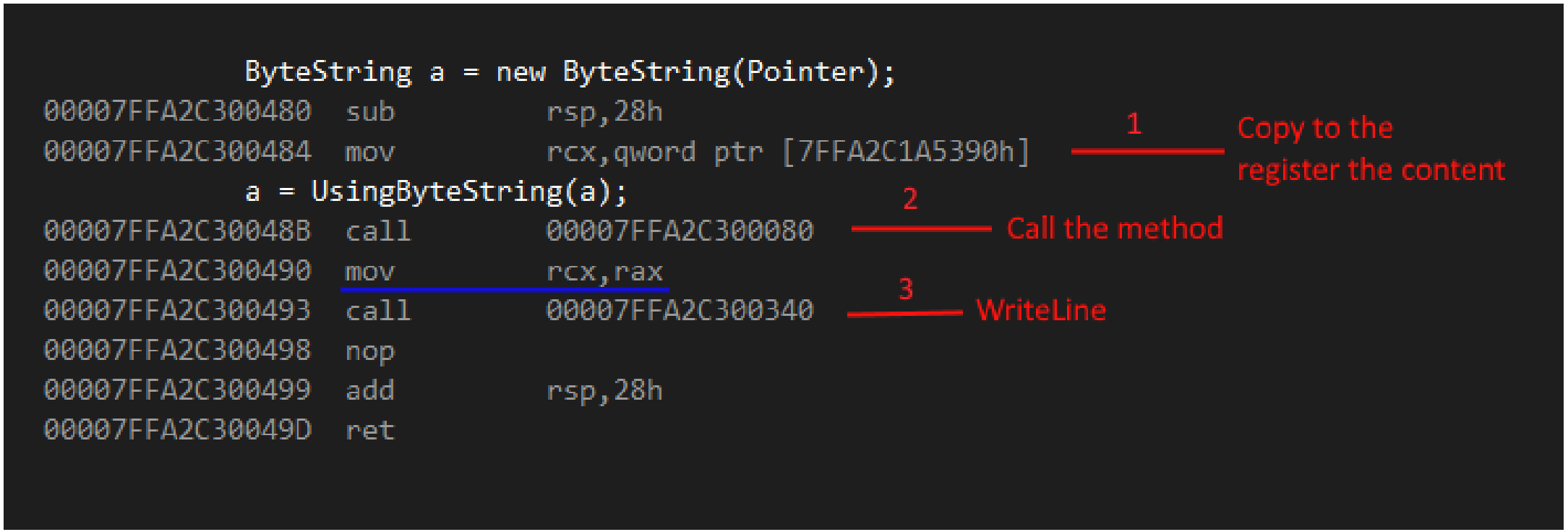
```
[MethodImpl(MethodImplOptions.NoInlining)]
public static ByteString UsingByteString(ByteString a)
{
    return a;
}

[MethodImpl(MethodImplOptions.NoInlining)]
public static long UsingLong(long a)
{
    return a;
}
```

```
public struct ByteString
{
    private byte* _pointer;

    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public ByteString(byte* ptr)
    {
        this._pointer = ptr;
    }

    public byte* Ptr
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get { return _pointer; }
    }
}
```

```
public static long Long;

static void Main(string[] args)
{
    long a = Long;
    a = UsingLong(a);

    Console.WriteLine(a);
}
```

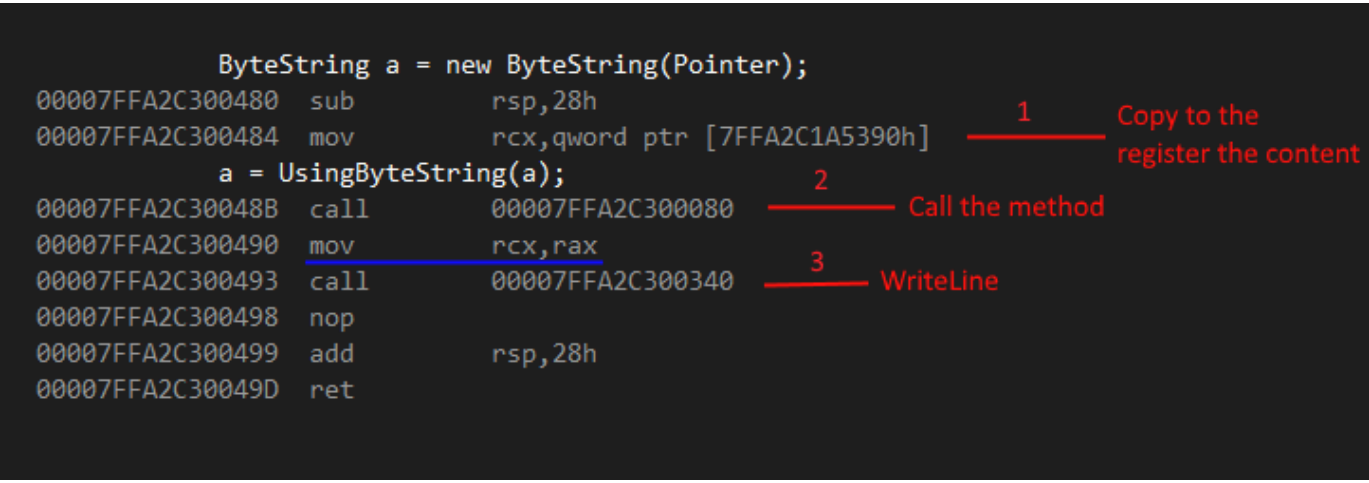


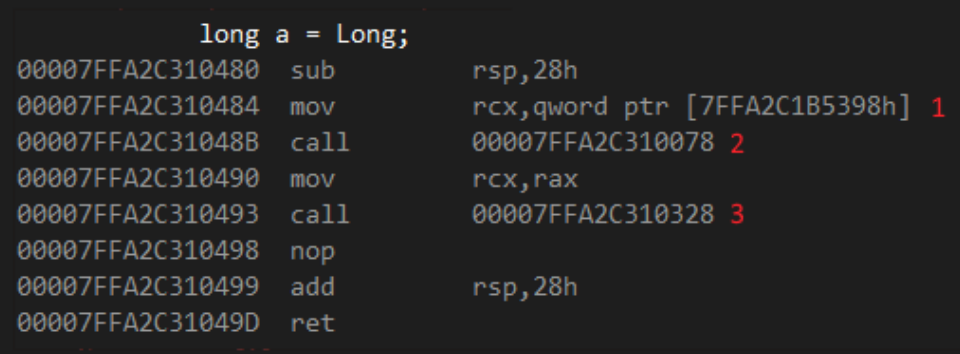
```
public static byte* Pointer;

static void Main(string[] args)
{
    ByteString a = new ByteString(Pointer);
    a = UsingByteString(a);

    Console.WriteLine((long)a.Ptr);
}
```



```
public static byte* Pointer;

static void Main(string[] args)
{
    ByteString a = new ByteString(Pointer);
    a = UsingByteString(a);

    Console.WriteLine((long)a.Ptr);
}
```

```
public static long Long;

static void Main(string[] args)
{
    long a = Long;
    a = UsingLong(a);

    Console.WriteLine(a);
}
```



I think you know where this is going…

# Wait for it…

# Generic Metaprogramming

```
public interface IMarker { }
public struct MarkerI : IMarker { }
public struct MarkerJ : IMarker { }
```

```
public static class Generic
{
    public static int Method<T>(int i, int j) where T : IMarker
    {
        if (typeof(T) == typeof(MarkerI))
        {
            return i;
        }

        return j;
    }
}
```

# Generic Metaprogramming

```
Generic.Method<MarkerI>(1,0)
```

```
public static class Generic
{
    public static int Method<MarkerI>(int i, int j) where T : IMarker
    {
        if (typeof(MarkerI) == typeof(MarkerI))
        {
            return i;
        }

        return j;
    }
}
```

# Generic Metaprogramming

```
Generic.Method<MarkerJ>(1,0)
```

```
public static class Generic
{
    public static int Method<MarkerJ>(int i, int j) where T : IMarker
    {
        if (typeof(MarkerJ) == typeof(MarkerI))
        {
            return i;
        }

        return j;
    }
}
```

# Why use Generic Metaprogramming?

- For those coming from a C++ background:
  - Same reasons apply.

- Avoid virtual calls on high frequency calls.
- Tailor algorithms based on usage.
- Exploit knowledge of constant values or simple expressions.
  - Improved & Tighter code generation.
- Avoid invariant conditionals.
  - CPU can predict the branch.
  - But, no code is always better than GREAT branch prediction.

# Zero Cost Extension Points

```
public class ObjectPool<T, TObjectLifecycle, TProcessAwareBehavior>
       where T : class
       where TObjectLifecycle : struct, IObjectLifecycle<T>
        where TProcessAwareBehavior : struct, IProcessAwareBehavior
{
       …
}
```

# Zero Cost Extension Points

```
public sealed class ObjectPool<T> : ObjectPool<T, FuncAllocator<T>,
                                    NonThreadAwareBehavior> where T : class
{
        public ObjectPool(Func<T> factory, int size = 16)
            : base(size, new FuncAllocator<T>(factory))
        { }
}
```

```
new ObjectPool<ObjectToPool>(() => new ObjectToPool());
```

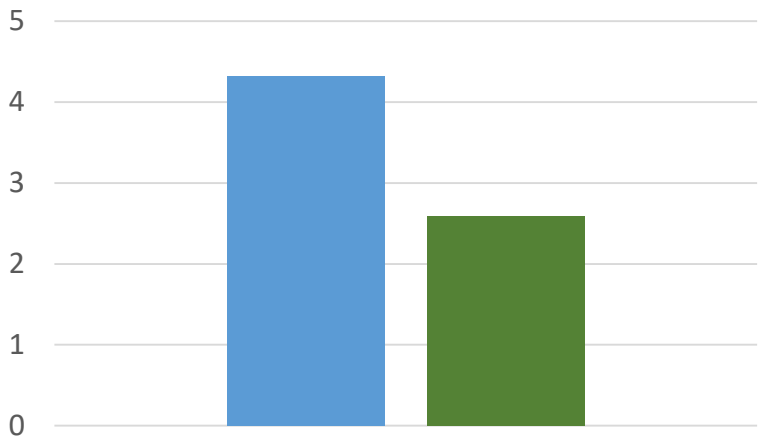
VS.

```
new ObjectPool<ObjectToPool, ObjectToPool.Allocator>();
```

```
public struct Allocator : IObjectLifecycle<ObjectToPool>
{
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public ObjectToPool New()
        {
                return new ObjectToPool();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(ObjectToPool value) { }
}
```

```
BenchmarkDotNet=v0.10.9, OS=Windows 10.0.16299
Processor=Intel Core i7-6700K CPU 4.00GHz (Skylake), ProcessorCount=8
Frequency=3914064 Hz, Resolution=255.4889 ns, Timer=TSC
  [Host]     : .NET Framework 4.7 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2556.0
  DefaultJob : .NET Framework 4.7 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2556.0
```

| | Method | Mean | Error | StdDev | InstructionRetired/Op |
|---|---|---|---|---|---|
| 1 | UsingFactory | 4.318 ns | 0.1533 ns | 0.1505 ns | 84 |
| 2 | UsingSpecificNew | 2.586 ns | 0.0153 ns | 0.0128 ns | 57 |



# Should I care about 27 instructions/op?

# Comparers

```
public class FastDictionary<TKey, TValue, TComparer>
                where TComparer : IEqualityComparer<TKey>
```

```
public sealed class Dictionary<TKey, TValue>
      : FastDictionary<TKey, TValue, IEqualityComparer<TKey>>
```

# Comparers

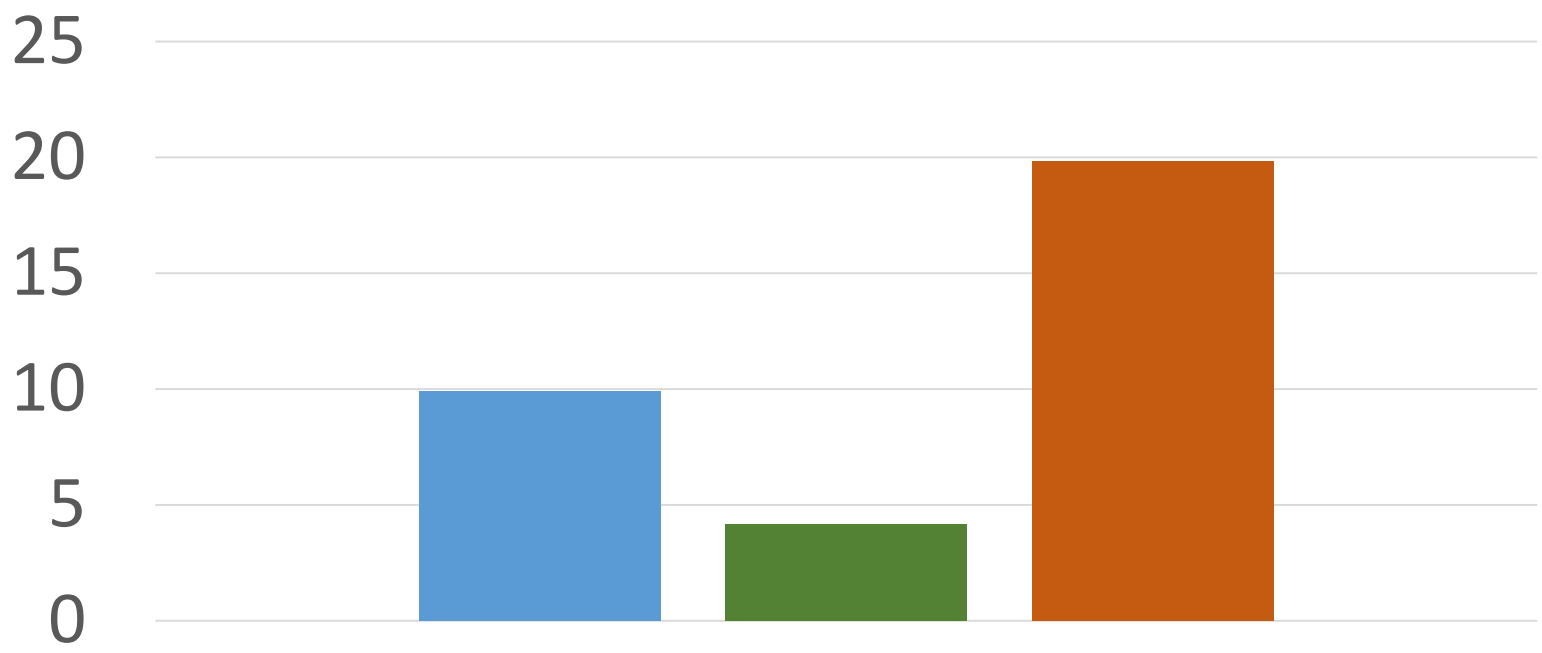
```
public struct NumericEqualityComparer : IEqualityComparer<long>
{
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Equals(long x, long y)
        {
            return x == y;
        }

        [MethodImpl (MethodImplOptions.AggressiveInlining)]
        public int GetHashCode(long obj)
        {
            …;
        }
}
```

1. `FastDictionary<long, long>(default(NumericEqualityComparer));`
2. `new FastDictionary<long, long, NumericEqualityComparer>();`
3. `new Dictionary<long, long>(default(NumericEqualityComparer));`

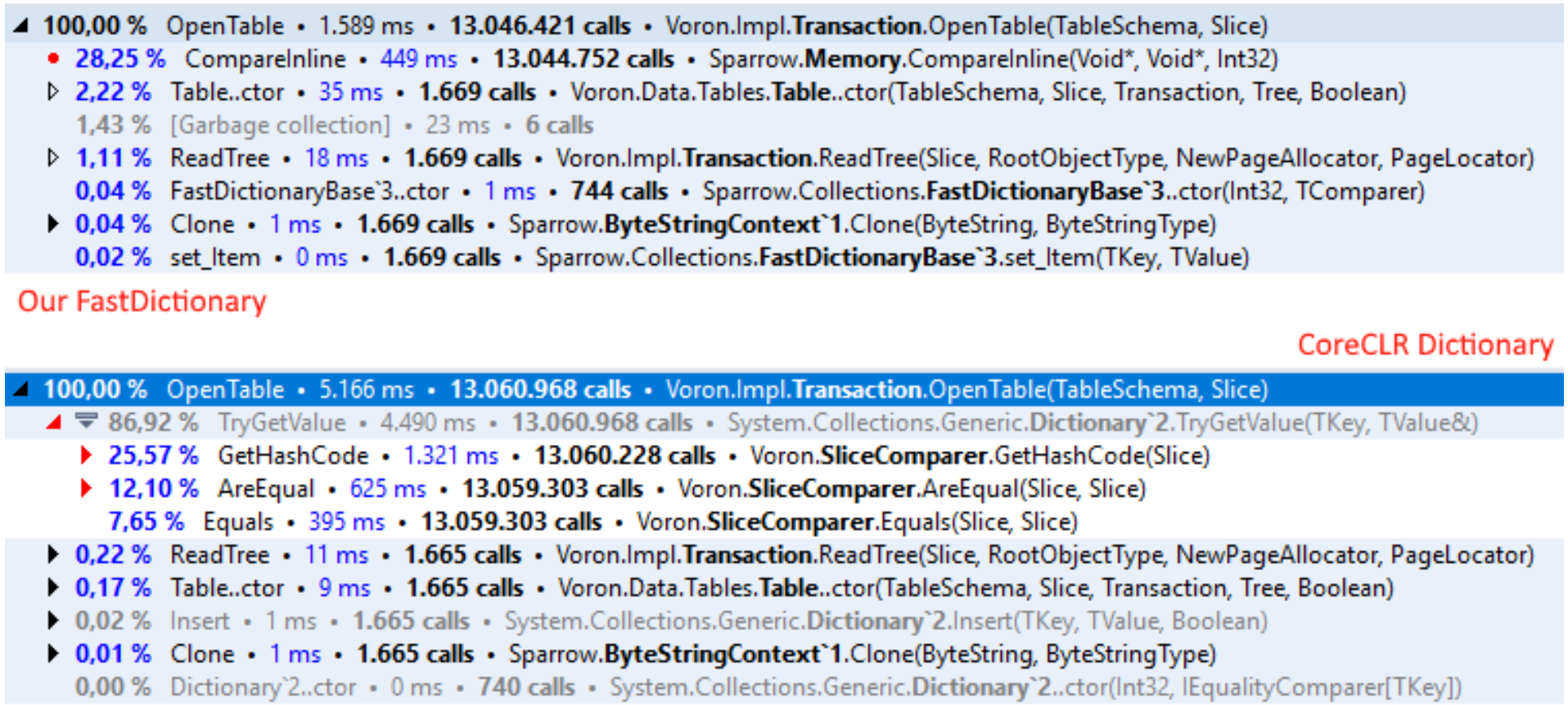
```
BenchmarkDotNet=v0.10.9, OS=Windows 10.0.16299
Processor=Intel Core i7-6700K CPU 4.00GHz (Skylake), ProcessorCount=8
Frequency=3914064 Hz, Resolution=255.4889 ns, Timer=TSC
  [Host]     : .NET Framework 4.7 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2556.0
  DefaultJob : .NET Framework 4.7 (CLR 4.0.30319.42000), 64bit RyuJIT-v4.7.2556.0
```

| | Method | Mean | Error | StdDev | InstructionRetired/Op |
|---|---|---|---|---|---|
| 1 | FastDictionaryIComparer | 9.894 us | 0.0182 us | 0.0152 us | 100715 |
| 2 | FastDictionarySpecific | 4.182 us | 0.0154 us | 0.0137 us | 70160 |
| 3 | Dictionary | 19.835 us | 0.0206 us | 0.0193 us | 164830 |

1. `FastDictionary<long, long>(default(NumericEqualityComparer));`
2. `new FastDictionary<long, long, NumericEqualityComparer>();`
3. `new Dictionary<long, long>(default(NumericEqualityComparer));`



# Comparers




1,589 ms

5,166 ms

That’s a 3.25x improvement on the wild

# Zero Cost Extension Points

```
public void Clear<TEviction>(TEviction evictionStrategy = default(TEviction))
    where TEviction : struct, IEviction<T>
{
    bool doDispose = typeof(IDisposable).IsAssignableFrom(typeof(T));

    if (typeof(TProcessAwareBehavior) == typeof(ThreadAwareBehavior))
    {
        for (int i = 0; i < _firstItems.Length; i++)
        {
            ref var bucket = ref _firstItems[i];
            if (doDispose)
                CacheAwareElement.ClearAndDispose(ref bucket, ref evictionStrategy);
            else
                CacheAwareElement.Clear(ref bucket, ref evictionStrategy);
        }
    }

    // Method continues here, but its uninteresting to us. ;)
}
```

JIT cannot know this is constant unless T is a struct

We cannot get rid of this branch so easily

# Zero Cost Extension Points

```
public void Clear<TEvictionStrategy>(TEviction evictionStrategy = default(TEviction))
    where TEviction : struct, IEviction<T>
{

    if (typeof(TProcessAwareBehavior) == typeof(ThreadAwareBehavior))
    {
        for (int i = 0; i < _firstItems.Length; i++)
        {
            ref var bucket = ref _firstItems[i];
            CacheAwareElement.Clear(ref bucket, ref evictionStrategy);
        }
    }

    // Method continues here, but it's uninteresting to us. ;)
}
```

```
public interface IEviction<in T>
{
    bool CanEvict(T item);
}

public struct AlwaysEvictStrategy<T> : IEviction<T>
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool CanEvict(T item)
    {
        return true;
    }
}

public struct NeverEvictStrategy<T> : IEviction<T>
{
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public bool CanEvict(T item)
    {
        return false;
    }
}
```

# Zero Cost Extension Points

```
public static void Clear<TEviction>(ref CacheAwareElement bucket,
                                    ref TEviction policy)
        where TEviction : struct, IEviction<T>
{
      T inst = bucket.Value1;
      if (inst != null && policy.CanEvict(inst))
      {
            Interlocked.CompareExchange(ref bucket.Value1, null, inst);
      }
}
```

# Zero Cost Extension Points

```
public static void Clear(ref CacheAwareElement bucket,
                         ref AlwaysEvictStrategy<T> policy)
{
    T inst = bucket.Value1;

    if (inst != null && true)
    {
        Interlocked.CompareExchange(ref bucket.Value1, null, inst);
    }
}
```

# Zero Cost Extension Points

```
public static void Clear(ref CacheAwareElement bucket,
                         ref AlwaysEvictStrategy<T> policy)
{
    T inst = bucket.Value1;

    if (inst != null)
    {
        Interlocked.CompareExchange(ref bucket.Value1, null, inst);
    }
}
                         // CanEvict will always evaluate to true.
                         // The code stays, the true gets optimized away
```

# Zero Cost Extension Points

```
public static void Clear(ref CacheAwareElement bucket,
                         ref NeverEvictStrategy<T> policy)
{
    T inst = bucket.Value1;

    if (inst != null && false)
    {
        Interlocked.CompareExchange(ref bucket.Value1, null, inst);
    }
}
```

# Zero Cost Extension Points

```
public static void Clear(ref CacheAwareElement bucket,
                         ref NeverEvictStrategy<T> policy)
{
    T inst = bucket.Value1;

    // JIT will discard the whole branch.
}
```

# Zero Cost Extension Points

```
public static void Clear(ref CacheAwareElement bucket,
                         ref NeverEvictStrategy<T> policy)
{
    // No reason to acquire a local variable
    // either if not going to be used ;)
}
```

```
public void Clear<TEvictionStrategy>(TEviction evictionStrategy = default(TEviction))
    where TEviction : struct, IEviction<T>
{

    if (typeof(TProcessAwareBehavior) == typeof(ThreadAwareBehavior))
    {
        for (int i = 0; i < _firstItems.Length; i++)
        {
            ref var bucket = ref _firstItems[i];
            CacheAwareElement.Clear(ref bucket, ref evictionStrategy);
        }
    }


    // Method continues here, but it's uninteresting to us. ;)
}
```

JIT wont optimize this loop out (YET!!)

# Code Specialization

```
public interface ILimitedOutputDirective { };
public struct NotLimited : ILimitedOutputDirective { };
public struct LimitedOutput : ILimitedOutputDirective { };

public interface ITableTypeDirective { };
public struct ByU32 : ITableTypeDirective { };
public struct ByU16 : ITableTypeDirective { };
```

```
private static int LZ4_compress_generic<TLimited, TTableType, TDictionaryType>(
            LZ4_stream_t_internal* dictPtr, byte* source, byte* dest,
            int inputSize, int maxOutputSize, int acceleration)
    where TLimited : struct, ILimitedOutputDirective
    where TTableType : struct, ITableTypeDirective
    where TDictionaryType : struct, IDictionaryTypeDirective
{
     ...
}
```

# Code Specialization

```
if (typeof(TTableType) == typeof(ByU16))
{
    ulong value = *((ulong*)forwardIp) * prime5bytes >> (40 - ByU16HashLog);
    forwardH = (int)(value & ByU16HashMask);
    ((ushort*)ctx->hashTable)[h] = (ushort)(ip - @base);
}
else if (typeof(TTableType) == typeof(ByU32))
{
    ulong value = (*((ulong*)forwardIp) * prime5bytes >> (40 - ByU32HashLog));
    forwardH = (int)(value & ByU32HashMask);
    ctx->hashTable[h] = (int)(ip - @base);
}
else throw new NotSupportedException("TTableType directive is not supported.");
```

# Zero Cost Façade

```
public unsafe interface IUnmanagedWriteBuffer : IDisposable
{
        int SizeInBytes { get; }
        void Write(byte[] buffer, int start, int count);
        void Write(byte* buffer, int length);
        void WriteByte(byte data);
        void EnsureSingleChunk(JsonParserState state);
        void EnsureSingleChunk(out byte* ptr, out int size);
}

public unsafe struct UnmanagedStreamBuffer : IUnmanagedWriteBuffer
{
    ...
}
public unsafe struct UnmanagedWriteBuffer : IUnmanagedWriteBuffer
{
   ...
}
```

# Zero Cost Façade

```
public sealed class BlittableWriter<TWriter> : IDisposable
        where TWriter : struct, IUnmanagedWriteBuffer
{
        private readonly JsonOperationContext _context;
        private readonly TWriter _unmanagedWriteBuffer;
        private AllocatedMemoryData _compressionBuffer;
        private AllocatedMemoryData _innerBuffer;
        private int _position;

        ...
}
```

# Zero Cost Façade

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void WriteNumber(int value, int sizeOfValue)
{
    // PERF: With the current JIT at 12 of January of 2017
            the switch statement don't get inlined.
    _unmanagedWriteBuffer.WriteByte((byte)value);
    if (sizeOfValue == sizeof(byte))
        return;

    _unmanagedWriteBuffer.WriteByte((byte)(value >> 8));
    if (sizeOfValue == sizeof(ushort))
        return;

    _unmanagedWriteBuffer.WriteByte((byte)(value >> 16));
    _unmanagedWriteBuffer.WriteByte((byte)(value >> 24));
}
```

# Hiding pointer/references under structs

```
public unsafe struct Page
{
    public readonly byte* Pointer;

    public Page(byte* pointer)
    {
        Pointer = pointer;
    }

    public long PageNumber
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        get { return ((PageHeader*)Pointer)->PageNumber; }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        set { ((PageHeader*)Pointer)->PageNumber = value; }
    }
}
```

# But is it really all this work worth the trouble?

3 weeks ago

100,00 % ReadObject • 517.225 ms • **6.523.208 calls** • Sparrow.Json.Parsing.**UnmanagedJsonParserHelper**.ReadObject(BlittableJsonDocumentBuilder, Stream, UnmanagedJsonParser, ManagedPinnedBuf...
- **77,26 %** Read • 399.618 ms • **6.641.138 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.Read
- **17,59 %** WriteDocumentMetadata • 90.981 ms • **6.523.208 calls** • Sparrow.Json.**BlittableWriter`1**.WriteDocumentMetadata(Int32, BlittableJsonToken)
- 3,38 % Read • 17.488 ms • **117.930 calls** • System.IO.Compression.**DeflateStream**.Read(Byte[], Int32, Int32)
- **1,32 %** ReadNestedObject • 6.851 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ReadNestedObject
- **0,06 %** FinalizeDocument • 327 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.FinalizeDocument

1 week ago

100,00 % ReadObject • 191.208 ms • **6.523.208 calls** • Sparrow.Json.Parsing.**UnmanagedJsonParserHelper**.ReadObject(BlittableJsonDocumentBuilder, Stream, UnmanagedJsonParser, ManagedPinnedBuffer)
- **70,08 %** Read • 133.995 ms • **6.641.138 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.Read
- **19,61 %** FinalizeDocument • 37.499 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.FinalizeDocument
- 8,21 % Read • 15.693 ms • **117.930 calls** • System.IO.Compression.**DeflateStream**.Read(Byte[], Int32, Int32)
- **0,54 %** ResetCaches • 1.032 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ResetCaches
- **0,33 %** ReadNestedObject • 628 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ReadNestedObject

today

100,00 % ReadObject • 130.133 ms • **6.523.208 calls** • Sparrow.Json.Parsing.**UnmanagedJsonParserHelper**.ReadObject(BlittableJsonDocumentBuilder, Stream, UnmanagedJsonParser, ManagedPinnedBuffer)
- **64,77 %** ReadInternal • 84.283 ms • **6.641.138 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ReadInternal
- **19,68 %** WriteDocumentMetadata • 25.611 ms • **6.523.208 calls** • Sparrow.Json.**BlittableWriter`1**.WriteDocumentMetadata(Int32, BlittableJsonToken)
- 13,17 % Read • 17.145 ms • **117.930 calls** • System.IO.Compression.**DeflateStream**.Read(Byte[], Int32, Int32)
- **0,85 %** ResetCaches • 1.108 ms • **6.523.208 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ResetCaches
- 0,16 % [Garbage collection] • 208 ms • **24 calls**

And we thought we were fast :)

today + 2 days

100,00 % ReadObject • 111.280 ms • **6.523.208 calls** • Sparrow.Json.Parsing.**UnmanagedJsonParserHelper**.ReadObject(BlittableJsonDocumentBuilder, Stream, Unmanage...
- **77,11 %** ReadInternal • 85.804 ms • **6.641.138 calls** • Sparrow.Json.**BlittableJsonDocumentBuilder**.ReadInternal
- 16,47 % Read • 18.329 ms • **117.930 calls** • System.IO.Compression.**DeflateStream**.Read(Byte[], Int32, Int32)
- **4,28 %** WriteDocumentMetadata • 4.767 ms • **6.523.208 calls** • Sparrow.Json.**BlittableWriter`1**.WriteDocumentMetadata(Int32, BlittableJsonToken)
- 0,36 % [Garbage collection] • 404 ms • **27 calls**
- **0,02 %** PushUnlikely • 23 ms • **923 calls** • Sparrow.Collections.**FastStack`1**.PushUnlikely(T)
- 0,01 % Read • 6 ms • **117.930 calls** • System.IO.Compression.**GZipStream**.Read(Byte[], Int32, Int32)

And I swear I wasnt even intending to improve this.
I was optimizing an entirely different process.

# That’s a 6.6x improvement on the wild

# Putting it all together

- Generic Metaprogramming over struct wrappers of types.
  - Allows to specialize generic code for specific conditions.
  - Allows to create zero-cost façade types.
    - Create interface based structs over the types.
    - Define generic methods over the interface type.
    - Profit. ☺
    - In Rust parlance: Poor man's traits (aka zero cost abstractions)
    - In C#: This "kinda" matches the shapes and extensions proposal
      - https://github.com/dotnet/csharplang/issues/164
  - Allows devirtualization to happen in difficult cases.

# Putting it all together

- Hiding pointers/references under structs allows:
  - Zero cost generics over pointers.
  - Struct types over sequences of preallocated data
    - Treat them as transparent indexes
    - Better memory locality.
    - Works over managed and unmanaged in the same way
  - Avoid allocation cost of lambda context capture.
    - When the possible outcomes are small.

# ONE FINAL THOUGHT

# Bottlenecks don’t disappear

## They just become more insidious

### Input-Output (3.0)

- BTree write-cliff [Bender]
- ~~Journal write dominates critical path~~
- ~~Single Write Lock~~

### CPU/Memory (3.0)

- JSON processing.
- ~~Allocations.~~
- ~~Virtual calls everywhere.~~
- Persistent Data Layout.
- Indexing ~~& Thread Scheduling~~

### Input-Output (4.0)

- BTree write-cliff. [Bender] (single node tera-scale)

### CPU/Memory (4.0)

- Thread state handoff.
- JSON processing (client only).
- Lucene.
- Algorithm memory locality.
- Asynchronous machinery.

Future Talks? Who knows…

# Thank you for coming!

# Want to know more?

## Start here!!!

RavenDB 4.0 Source Code: Grep my commits (author: redknightlois) ☺
https://github.com/ravendb/ravendb/search?o=desc&q=author%3Aredknightlois&s=author-date&type=Commits
Local (arena) Memory Allocators - John Lakos [ACCU 2017]
https://www.youtube.com/watch?v=d1DpVR0tw0U
When a Microsecond is an Eternity: High Performance Trading Systems in C++ - Carl Cook [CppCon 2017]
https://www.youtube.com/watch?v=NH1Tta7purM
An Overview of Program Optimization Techniques - Mathias Gaunard [ACCU 2017]
https://www.youtube.com/watch?v=pEvm5NNc6ko
MIT 6.172 Performance Engineering of Software Systems, Fall 2010 (BTree Write Cliff)
https://www.youtube.com/watch?v=9Rb85cOXTKU
Understanding Flash: The Write Cliff (SSDs)
https://flashdba.com/2014/11/24/understanding-flash-the-write-cliff/
Oren Eini's blog – mostly database tech (I write occasionally as guest writer)
https://ayende.com/blog/
CoreCLR issues on performance and optimization
https://github.com/dotnet/coreclr/issues?utf8=✓&q=label%3Atenet-performance%20%20label%3Aoptimization%20
BenchmarkDotNet
https://github.com/dotnet/BenchmarkDotNet