corvalius

Metaprogramming “for the masses”

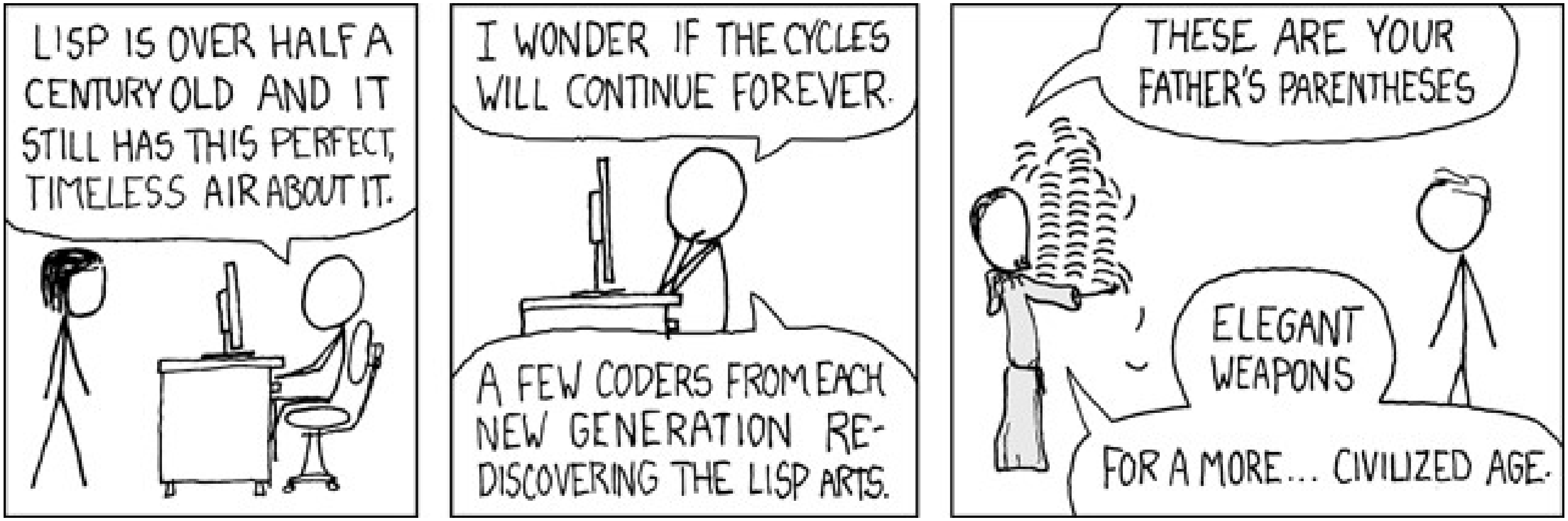
Federico Lois
Twitter: @federicolois
Github: redknightlois
Repo: metaprogramming

```
template< bool b >
struct algorithm_selector {
  template< typename T >
  static void implementation( T& object )
  {

  }
};
```

First. Some FACTS

Metaprogramming ain’t a new concept


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

Source: https://xkcd.com/297/

C# was not designed to do this kind of stuff…

But the runtime is…

It’s hard!!!

HARDCORE

H

Assembly

Unintuitive code

Wacky C#

DOTNEXT CONTENT RATING

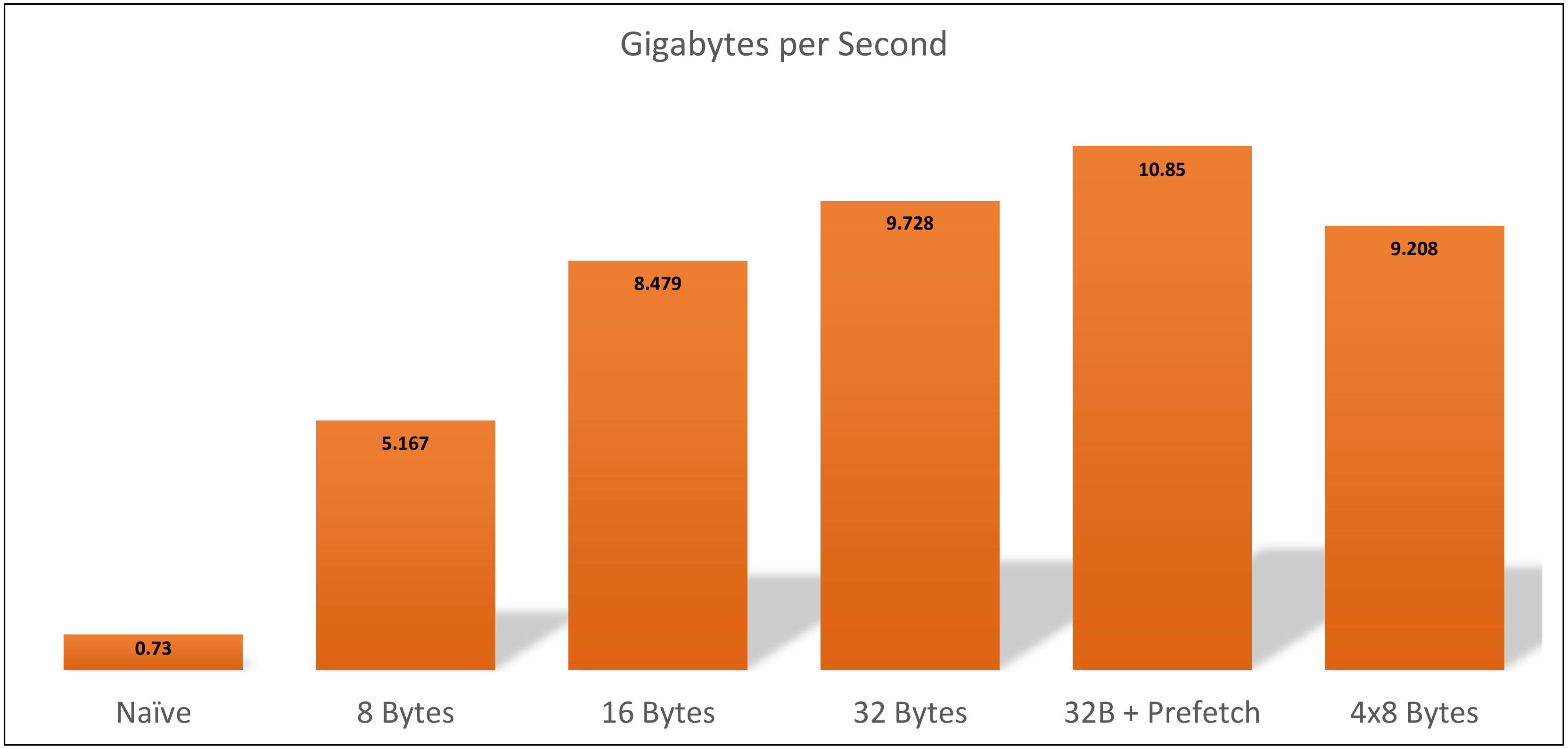
Why we should ♡ metaprogramming

- Allows us to define, reusable behavior
- The code generated is relatively fast
  - We are not there yet to claim SOTA speed
- 80% of the speed for 20% of the effort
- JIT engineers ‘love’ we raise weird code-gen issues
  - As it should be ☺

We are not forced into inheritance and composition for designing reusable code anymore

A quick rehash from earlier talks.

Do you remember these slides?



```
public interface IEvictionStrategy<in T> where T : class
{
    bool CanEvict(T item);
}

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

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

Today we will focus on…

Image: @joel_herzog

Example 1: Binary Tree Traversal

Technique: Method Strategy

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public partial class Tree
{
    public Node Root { get; private set; }

    private Node Insert(Node root, int v)
    {
        if (root == null)
        {
            root = new Node();
            root.Value = v;
        }
        else if (v < root.Value)
        {
            root.Left = Insert(root.Left, v);
        }
        else
        {
            root.Right = Insert(root.Right, v);
        }

        return root;
    }
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public partial class Tree
{
    public void Traverse(Node root)
    {
        if (root == null)
            return;

        Traverse(root.Left);
        Traverse(root.Right);
    }
}
```

BOOOORING!!!!

Let’s do it the ‘yield’ way…

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    IEnumerable<Node> Enumerate(Node node);
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public IEnumerable<Node> Enumerate(Node root)
    {
        if (root == null)
            yield break;

        yield return root.Left;
        yield return root.Right;
    }
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    IEnumerable<Node> Enumerate(Node node);
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public IEnumerable<Node> Enumerate(Node root)
    {
        if (root == null)
            yield break;

        yield return root.Left;
        yield return root.Right;
    }
}
```

```
public void Traverse<TTraverseStrategy>(Node node)
    where TTraverseStrategy : struct, ITraverseStrategy
{
    TTraverseStrategy strategy = default;
    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public void Traverse<TTraverseStrategy>(Node node)

    where TTraverseStrategy : struct, ITraverseStrategy

{
    TTraverseStrategy strategy = default;

    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

This is our strategy type

```
public void Traverse<TTraverseStrategy>(Node node)

    where TTraverseStrategy : struct, ITraverseStrategy

{
    TTraverseStrategy strategy = default;

    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

This is our strategy type

```
public void Traverse<TTraverseStrategy>(Node node)

    where TTraverseStrategy : struct, ITraverseStrategy
```

The struct ensures we don’t need a reference

```
{
    TTraverseStrategy strategy = default;

    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

This is our strategy

```
public void Traverse<TTraverseStrategy>(Node node)

    where TTraverseStrategy : struct, ITraverseStrategy
```

The struct ensures we don't need a reference

```
{
    TTraverseStrategy strategy = default;
```

This is a pseudo instantiation.

```
    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

This is our strategy

```
public void Traverse<TTraverseStrategy>(Node node)

    where TTraverseStrategy : struct, ITraverseStrategy

{
    TTraverseStrategy strategy = default;

    foreach (var n in strategy.Enumerate(node))
        Traverse<TTraverseStrategy>(n);
}
```

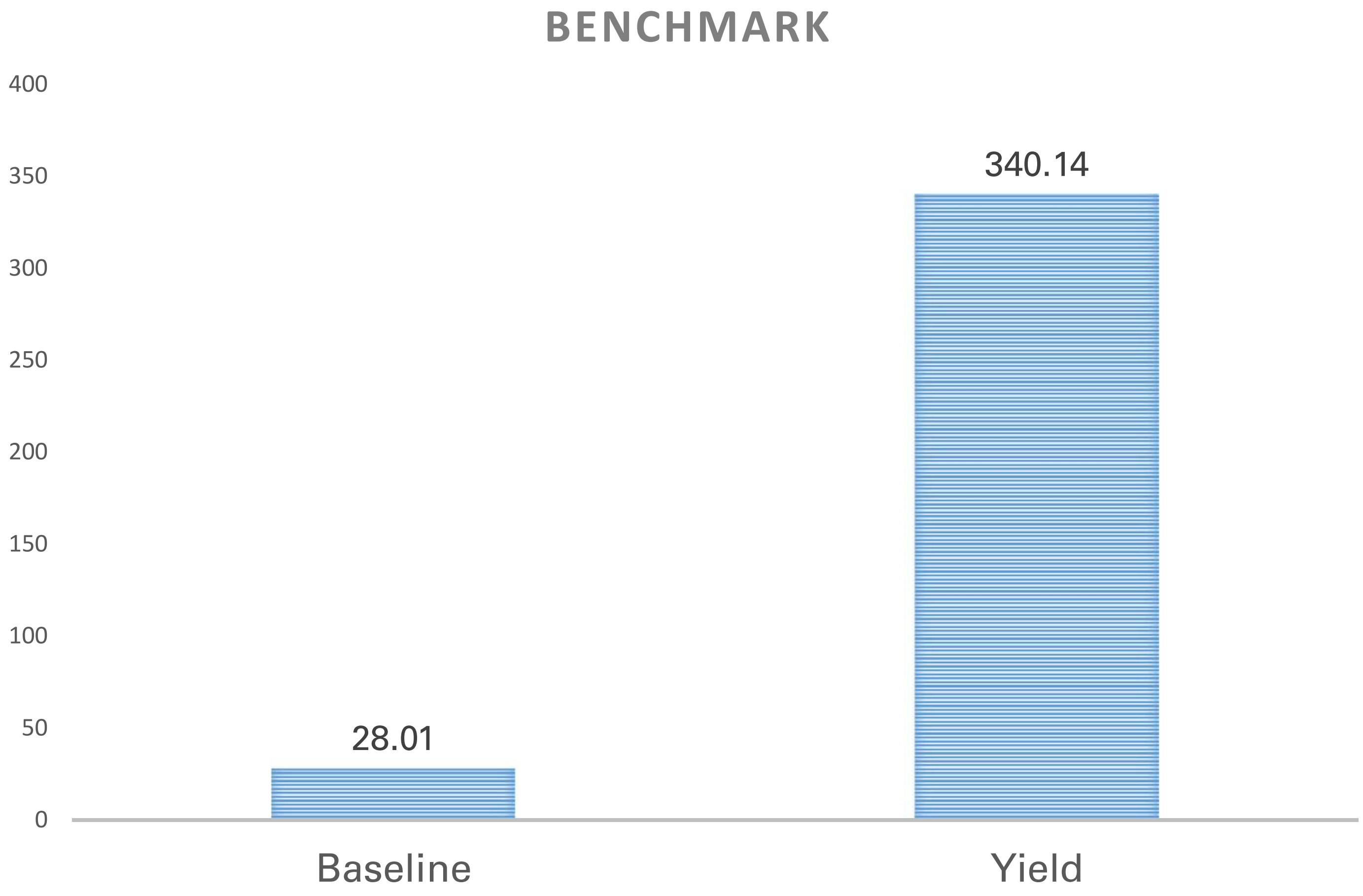
The struct ensures we don’t need a reference

This is a pseudo instantiation.

The recursive call

Fight (??)

All results in μs



```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    IEnumerable<Node> Enumerate(Node node);
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public IEnumerable<Node> Enumerate(Node root)
    {
        if (root == null)
            yield break;

        yield return root.Left;
        yield return root.Right;
    }
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy;
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public IEnumerable<Node> Enumerate(Node root)
    {
        if (root == null)
            yield break;

        yield return root.Left;
        yield return root.Right;
    }
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy;
}
```

```
public void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy
{
    if (node == null)
        return;

    TTraverseStrategy strategy = default;
    strategy.Traverse<TTraverseStrategy>(node.Left);
    strategy.Traverse<TTraverseStrategy>(node.Right);
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

```
public interface ITraverseStrategy
{
    void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy;
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy
    {
        if (node == null)
            return;

        TTraverseStrategy strategy = default;
        strategy.Traverse<TTraverseStrategy>(node.Left);
        strategy.Traverse<TTraverseStrategy>(node.Right);
    }
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

```
public class Node
{
    public int Value;
    public Node Left;
    public Node Right;
}
```

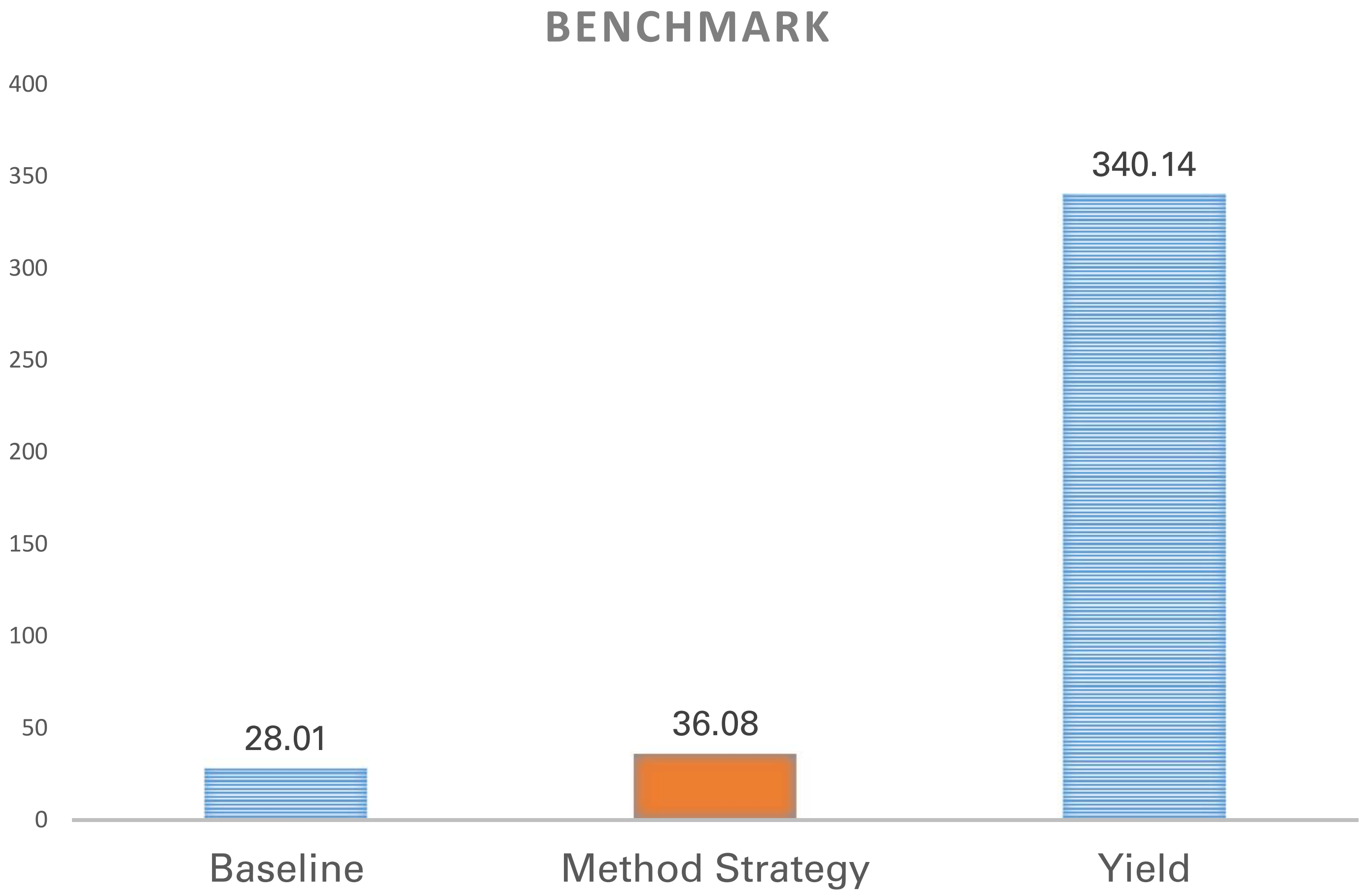
```
public interface ITraverseStrategy
{
    void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy;
}
```

```
public struct InfixStrategy : ITraverseStrategy
{
    public void Traverse<TTraverseStrategy>(Node node)
         where TTraverseStrategy : struct, ITraverseStrategy
    {
        if (node == null)
            return;

        TTraverseStrategy strategy = default;
        strategy.Traverse<TTraverseStrategy>(node.Left);
        strategy.Traverse<TTraverseStrategy>(node.Right);
    }
}
```

```
tree.Traverse<InfixStrategy>(tree.Root);
```

All results in μs



Example 2: Exploratory Unrolling

Technique: Code Pruning

```
for (int i = 0; i < _floatArray.Length; i++)
    _floatArray[i] = i;
```

Assume: _floatArray.Length % 8 == 0

Expect: No corner cases

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
for (int i = 0; i < _floatArray.Length; i += 8)
{
    // Each one of the accesses look like this...
    //     cmp         eax,r8d
    //     jae         00007FF7DF332623
    //     movsxd      r9,eax
    //     vxorps      xmm0,xmm0,xmm0
    //     vcvtsi2ss   xmm0,xmm0,eax
    //     vmovss      dword ptr[rcx + r9 * 4 + 10h], xmm0
    _floatArray[i] = i;

    _floatArray[i + 1] = i + 1;
    _floatArray[i + 2] = i + 2;
    _floatArray[i + 3] = i + 3;
    _floatArray[i + 4] = i + 4;
    _floatArray[i + 5] = i + 5;
    _floatArray[i + 6] = i + 6;
    _floatArray[i + 7] = i + 7;
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
for (int i = 0; i < _floatArray.Length; i += 8)
{
    // Each one of the accesses look like this...
    //     cmp         eax,r8d
    //     jae         00007FF7DF332623
    //     movsxd      r9,eax
    //     vxorps      xmm0,xmm0,xmm0
    //     vcvtsi2ss   xmm0,xmm0,eax
    //     vmovss      dword ptr[rcx + r9 * 4 + 10h], xmm0
    _floatArray[i] = i;

    _floatArray[i + 1] = i + 1;
    _floatArray[i + 2] = i + 2;
    _floatArray[i + 3] = i + 3;
    _floatArray[i + 4] = i + 4;
    _floatArray[i + 5] = i + 5;
    _floatArray[i + 6] = i + 6;
    _floatArray[i + 7] = i + 7;
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
for (int i = 0; i < _floatArray.Length; i += 8)
{
    // Each one of the accesses look like this...
    //      cmp         eax,r8d
    //      jae         00007FF7DF332623
    //      movsxd      r9,eax
    //      vxorps      xmm0,xmm0,xmm0
    //      vcvtsi2ss   xmm0,xmm0,eax
    //      vmovss      dword ptr[rcx + r9 * 4 + 10h], xmm0
    _floatArray[i] = i;

    _floatArray[i + 1] = i + 1;

    _floatArray[i + 2] = i + 2;

    . . .

    _floatArray[i + 6] = i + 6;

    _floatArray[i + 7] = i + 7;
}
```

```
public interface IUnrollConfiguration<T>
{
    int Step { get; }
    void Act(int index, ref T data);
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
for (int i = 0; i < _floatArray.Length; i += 8)
{
    // Each one of the accesses look like this...
    //     cmp         eax,r8d
    //     jae         00007FF7DF332623
    //     movsxd      r9,eax
    //     vxorps      xmm0,xmm0,xmm0
    //     vcvtsi2ss   xmm0,xmm0,eax
    //     vmovss      dword ptr[rcx + r9 * 4 + 10h], xmm0
    _floatArray[i] = i;

    _floatArray[i + 1] = i + 1;

    _floatArray[i + 2] = i + 2;

    . . .

    _floatArray[i + 6] = i + 6;

    _floatArray[i + 7] = i + 7;
}
```

The implicit step

```
public interface IUnrollConfiguration<T>
{
    int Step { get; }
    void Act(int index, ref T data);
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
for (int i = 0; i < _floatArray.Length; i += 8)
{
    // Each one of the accesses look like this...
    //      cmp         eax,r8d
    //      jae         00007FF7DF332623
    //      movsxd      r9,eax
    //      vxorps      xmm0,xmm0,xmm0
    //      vcvtsi2ss   xmm0,xmm0,eax
    //      vmovss      dword ptr[rcx + r9 * 4 + 10h], xmm0
    _floatArray[i] = i;

    _floatArray[i + 1] = i + 1;

    _floatArray[i + 2] = i + 2;

    . . .

    _floatArray[i + 6] = i + 6;

    _floatArray[i + 7] = i + 7;
}
```

The implicit step

```
public interface IUnrollConfiguration<T>
{
    int Step { get; }
    void Act(int index, ref T data);
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
    _floatArray[i] = i;
```

```
public struct UnrollAction2 : IUnrollConfiguration<float>
{
    public int Step => 2;

    public void Act(int index, ref float data)
    {
        data = index;
    }
}
```

```
ExecuteUnrolled<UnrollAction2, float>(_floatArray);
```

So how do we write this?

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
private void ExecuteUnrolled<TUnroller, T>(T[] data)
        where TUnroller : struct, IUnrollConfiguration<T>
{
    TUnroller unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
private void ExecuteUnrolled<TUnroller, T>(T[] data)
       where TUnroller : struct, IUnrollConfiguration<T>
 {
    TUnroller unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
private void ExecuteUnrolled<TUnroller, T>(T[] data)
       where TUnroller : struct, IUnrollConfiguration<T>
{
    TUnroller unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller 
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
private void ExecuteUnrolled<TUnroller, T>(T[] data)
        where TUnroller : struct, IUnrollConfiguration<T>
{
    TUnroller unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

```
private void ExecuteUnrolled<UnrollAction2, float>(float[] data)
{
    UnrollAction2 unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

```
for (int i = 0; i < _floatArray.Length; i++)
   _floatArray[i] = i;
```

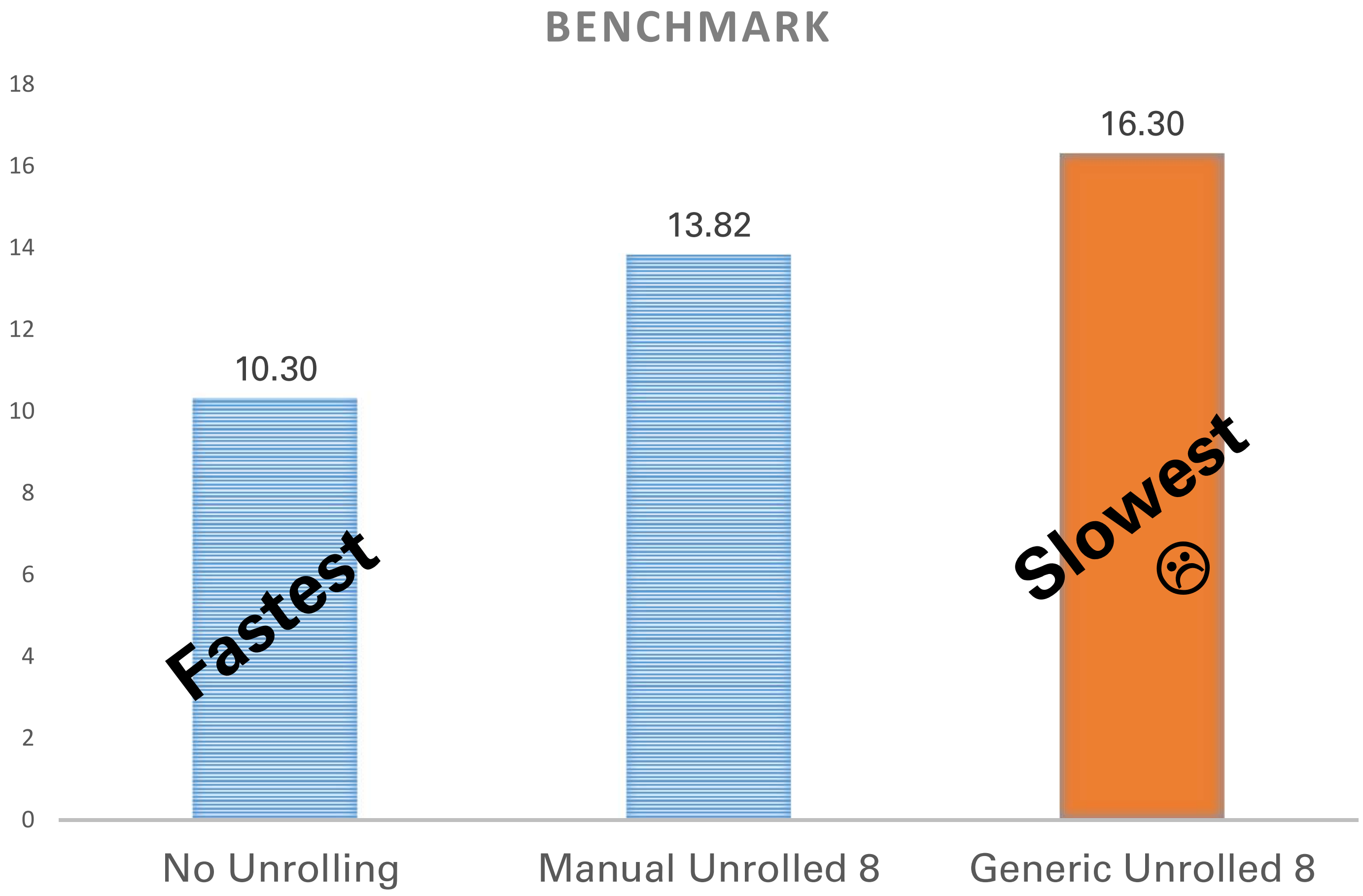
```
private void ExecuteUnrolled<UnrollAction2, float>(float[] data)
{
    UnrollAction2 unroller = default;
    if (unroller.Step > 8)
        throw new NotImplementedException("The unroller
implementation doesnt support chunks bigger than 8");

    for (int i = 0; i < data.Length; i += unroller.Step)
    {
        unroller.Act(i, ref data[i]);
        if (unroller.Step == 1) continue;

        unroller.Act(i + 1, ref data[i + 1]);
        if (unroller.Step == 2) continue;

        ...
        unroller.Act(i + 7, ref data[i + 7]);
    }
}
```

All results in ms



Fastest

Slowest ☹

Bound Checks ☺

Example 3: Matrix Accessors

Technique: Interface Devirtualization

```
public class Matrix
{
    protected readonly int _xSize;
    protected readonly int _ySize;
    protected float[] _storage;
}


public Matrix(int xSize, int ySize)
{
    this._xSize = xSize;
    this._ySize = ySize;
    _storage = new float[xSize * ySize];
}
```

```
public float this[int x, int y]
{
    get
    {
        int idx = x * _ySize + y;
        return _storage[idx];
    }
    set
    {
        int idx = x * _ySize + y;
        _storage[idx] = value;
    }
}
```

```
public class Matrix
{
    protected readonly int _xSize;
    protected readonly int _ySize;
    protected float[] _storage;
}


public Matrix(int xSize, int ySize)
{
    this._xSize = xSize;
    this._ySize = ySize;
    _storage = new float[xSize * ySize];
}
```

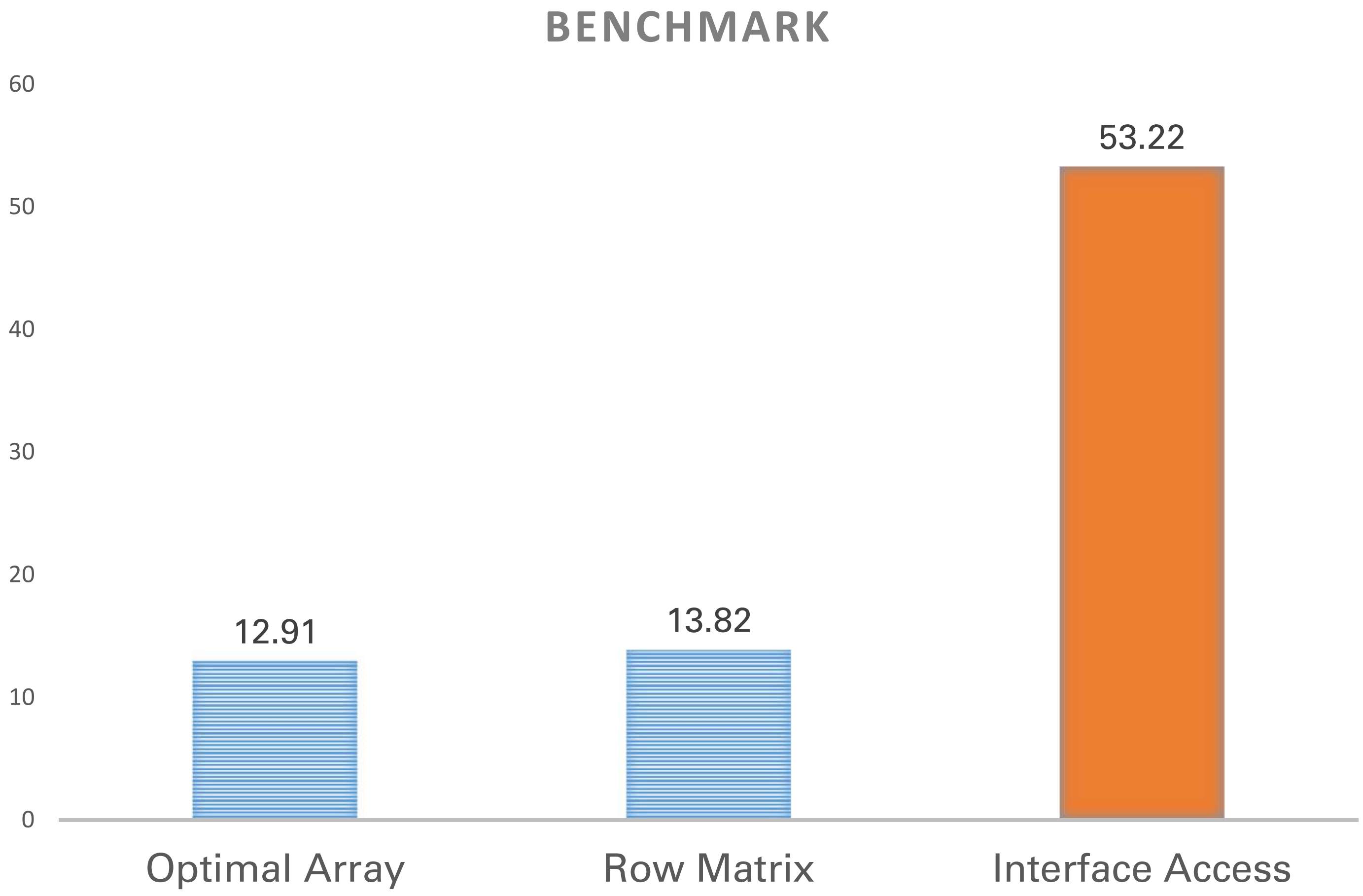
```
public float this[int x, int y]
{
    get
    {
        int idx = x * _ySize + y;
        return _storage[idx];
    }
    set
    {
        int idx = x * _ySize + y;
        _storage[idx] = value;
    }
}
```

```
public class RowMatrix
{
    IStorageLayout<float> _storage;

    public RowMatrix(int xSize, int ySize)
    {
        _storage = new RowFirst<float>();
        _storage.Initialize(xSize, ySize);
    }

    public float this[int x, int y]
    {
        get { return _storage.Get(x, y); }
        set { _storage.Set(x, y, value); }
    }
}
```

All results in ms



```
public class Matrix<TStorage, T>
    where TStorage : struct, IStorageLayout<T>
{
    TStorage _storage;

    public Matrix(int xSize, int ySize)
    {
        _storage.Initialize(xSize, ySize);
    }

    public T this[int x, int y]
    {
        get { return _storage.Get(x, y); }
        set { _storage.Set(x, y, value); }
    }
}
```

```
public class Matrix<TStorage, T>
    where TStorage : struct, IStorageLayout<T>
{
    TStorage _storage;

    public Matrix(int xSize, int ySize)
    {
        _storage.Initialize(xSize, ySize);
    }

    public T this[int x, int y]
    {
        get { return _storage.Get(x, y); }
        set { _storage.Set(x, y, value); }
    }
}
```

```
public interface IStorageLayout<T>
{
    void Initialize(int x, int y);
    void Set(int x, int y, T value);
    T Get(int x, int y);
}
```

```
public class Matrix<TStorage, T>
    where TStorage : struct, IStorageLayout<T>
{
    TStorage _storage;

    public Matrix(int xSize, int ySize)
    {
        _storage.Initialize(xSize, ySize);
    }

    public T this[int x, int y]
    {
        get { return _storage.Get(x, y); }
        set { _storage.Set(x, y, value); }
    }
}
```

```
public interface IStorageLayout<T>
{
    void Initialize(int x, int y);
    void Set(int x, int y, T value);
    T Get(int x, int y);
}
```

```
public class Matrix<TStorage, T>
    where TStorage : struct, IStorageLayout<T>
```

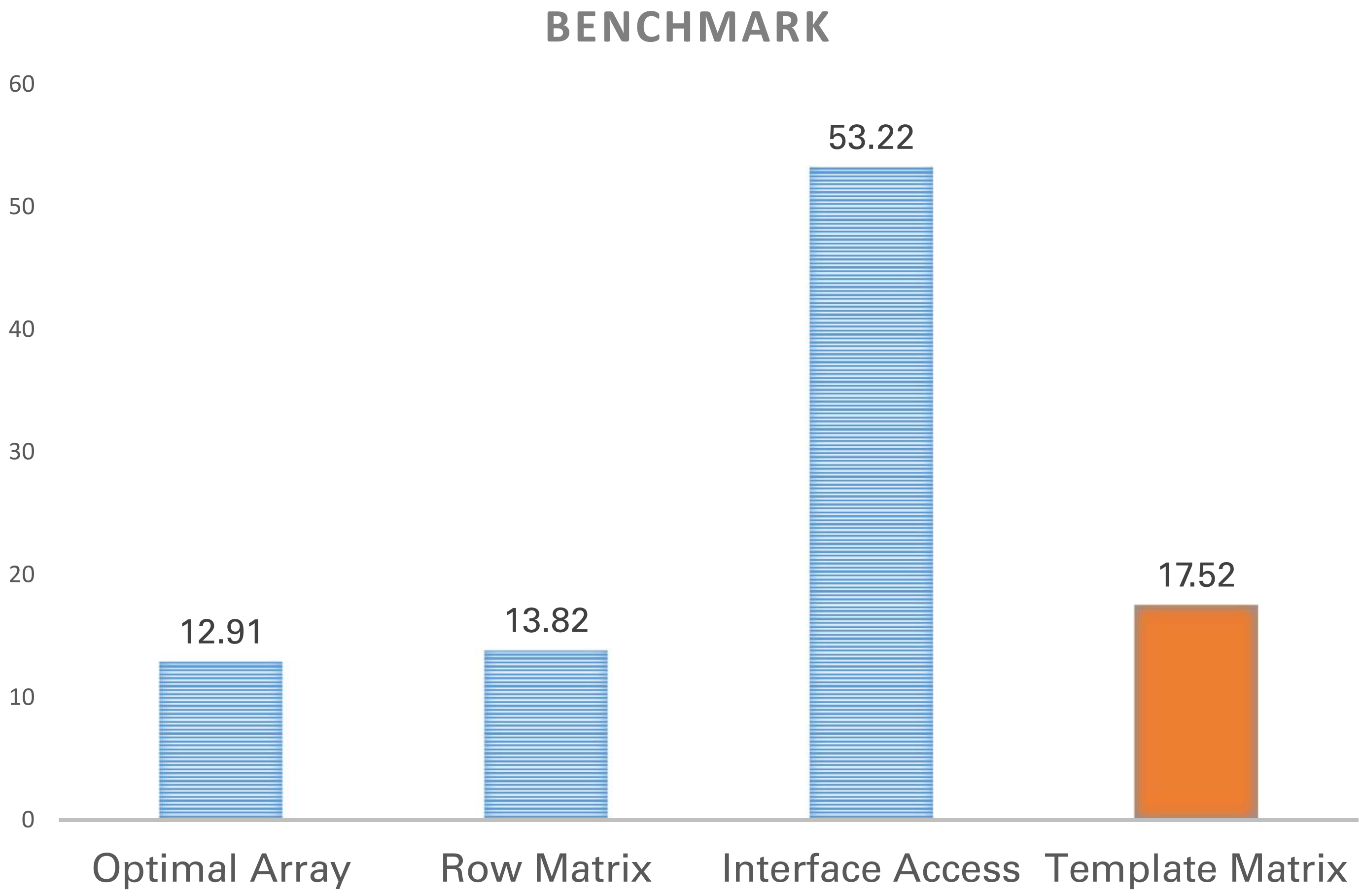
```
public interface IStorageLayout<T>
{
    void Initialize(int x, int y);
    void Set(int x, int y, T value);
    T Get(int x, int y);
}
```

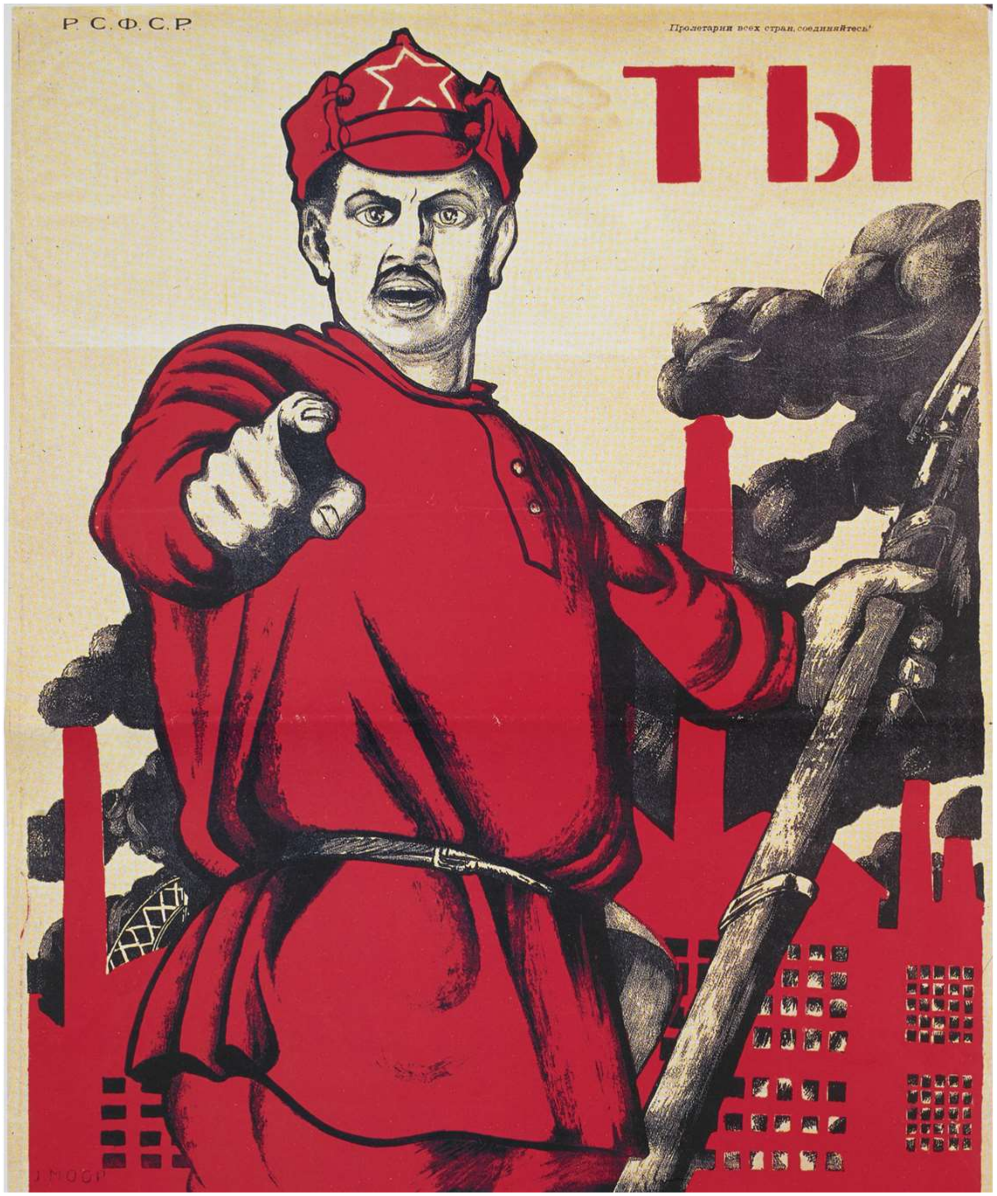
Not what we want to write

```
var m = new Matrix<RowStorage<float>>(…);
var m = new Matrix<RowStorage<float>>(…);
```

```
public class Matrix<TStorage, T>
    where T : TStorage<implicit T>
    where TStorage : struct, IStorageLayout<T>
```

All results in ms




Р.С.Ф.С.Р.
Пролетарии всех стран, соединяйтесь!
ТЫ
Д. Моор

Want to know more?

Start here!!!

Allocators experimental branch (author: redknightlois) ☺
https://github.com/Corvalius/ravendb/tree/allocators/src/Sparrow ← Allocator.*.cs files
What Every Programmer Should Know About Memory – Ulrich Drepper
https://www.akkadia.org/drepper/cpumemory.pdf
Going Nowhere Faster – Chandler Carruth [CppCon 2017]
https://www.youtube.com/watch?v=2EWejmkKlxs
Beating CoreCLR's own C++ code with CoreCLR 3.0 [DotNext 2019 Moscow]
 - if you didn't attend I suggest to watch it when video is available
Grace Hopper – Nanoseconds
https://www.youtube.com/watch?v=JEpsKnWZrJ8
https://www.youtube.com/watch?v=ZR0ujwlvbkQ (whole lecture – worth it)
BenchmarkDotNet
https://github.com/dotnet/BenchmarkDotNet

Before you leave

- Don’t use this techniques blindly
  - Always think them as tools to achieve goals
- The balance between performance and maintenability is key to success

Thank you all for coming!

H HARDCORE

ASSEMBLY, UNINTUITIVE CODE, WACKY C#