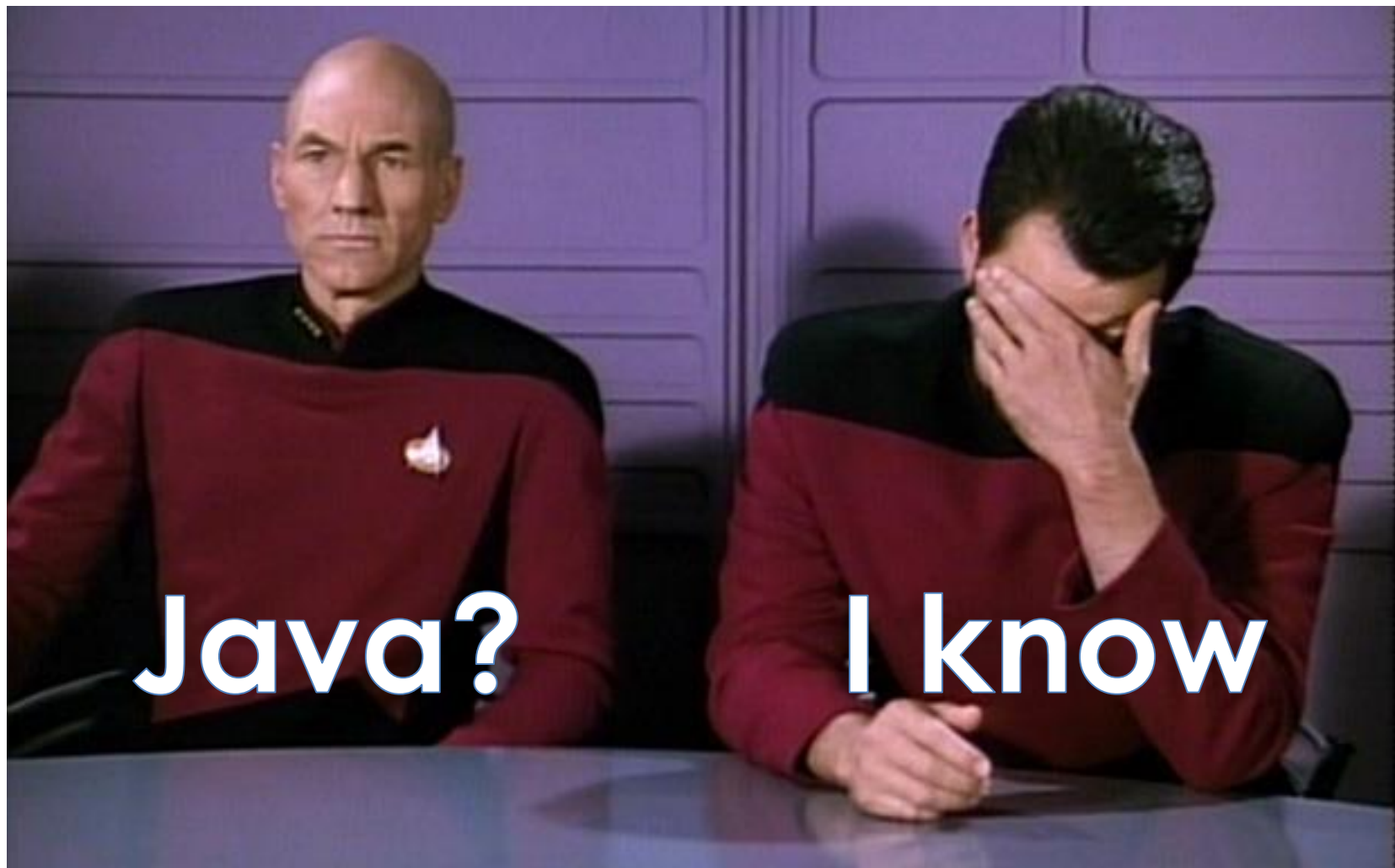


High Performance Managed Languages

Martin Thompson - @mjpt777

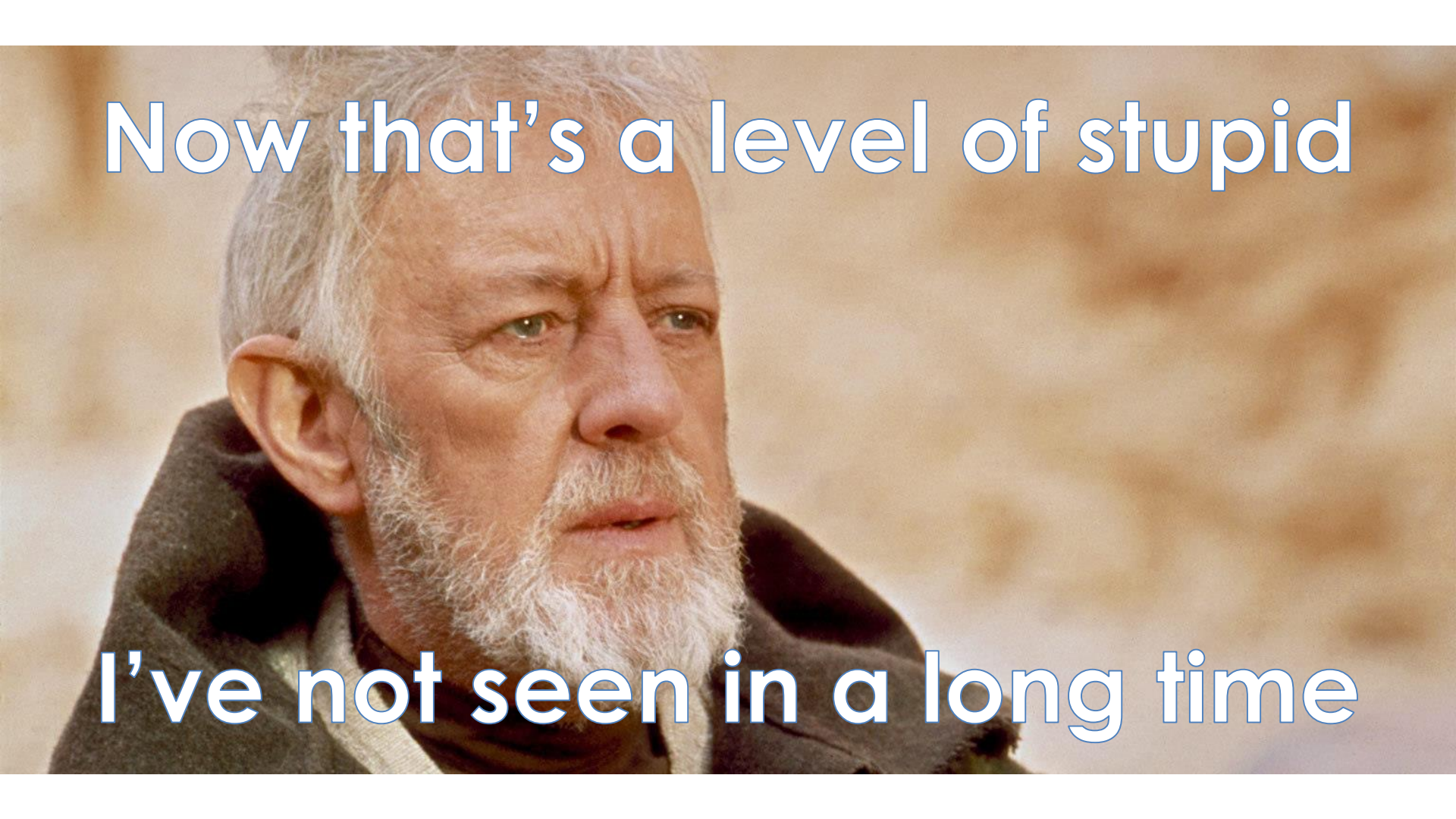
**Really, what is your preferred
platform for building HFT
applications?**



Java?

I know

**Why do you build low-latency
applications on a
GC'ed platform?**



Now that's a level of stupid

I've not seen in a long time

Agenda

- 1. Let's set some Context**
- 2. Runtime Optimisation**
- 3. Garbage Collection**
- 4. Algorithms & Design**

Some Context

Let's be clear

**A Managed Runtime is not
always the best choice...**

Latency Arbitrage?

Altera®

FPGA



Two questions...

**Why build on a
Managed Runtime?**

**Can managed languages
provide good performance?**

**We need to follow the
evidence...**

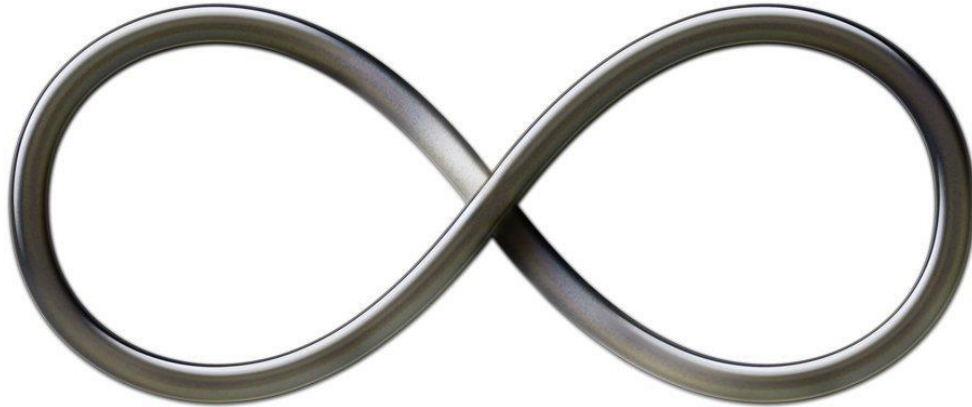
“You investigate
for curiosity,
because it is
unknown, not
because you know
the answer.”

Richard Feynman



Are native languages faster?

Time?



Skills & Resources?

**What can, or should, be
outsourced?**

CPU vs Memory Performance

**How much time to perform an
addition operation on 2 integers?**

1 CPU Cycle
< 1ns

Sequential Access

-

**Average time in ns/op to sum all
longs in a 1GB array?**

Access Pattern Benchmark

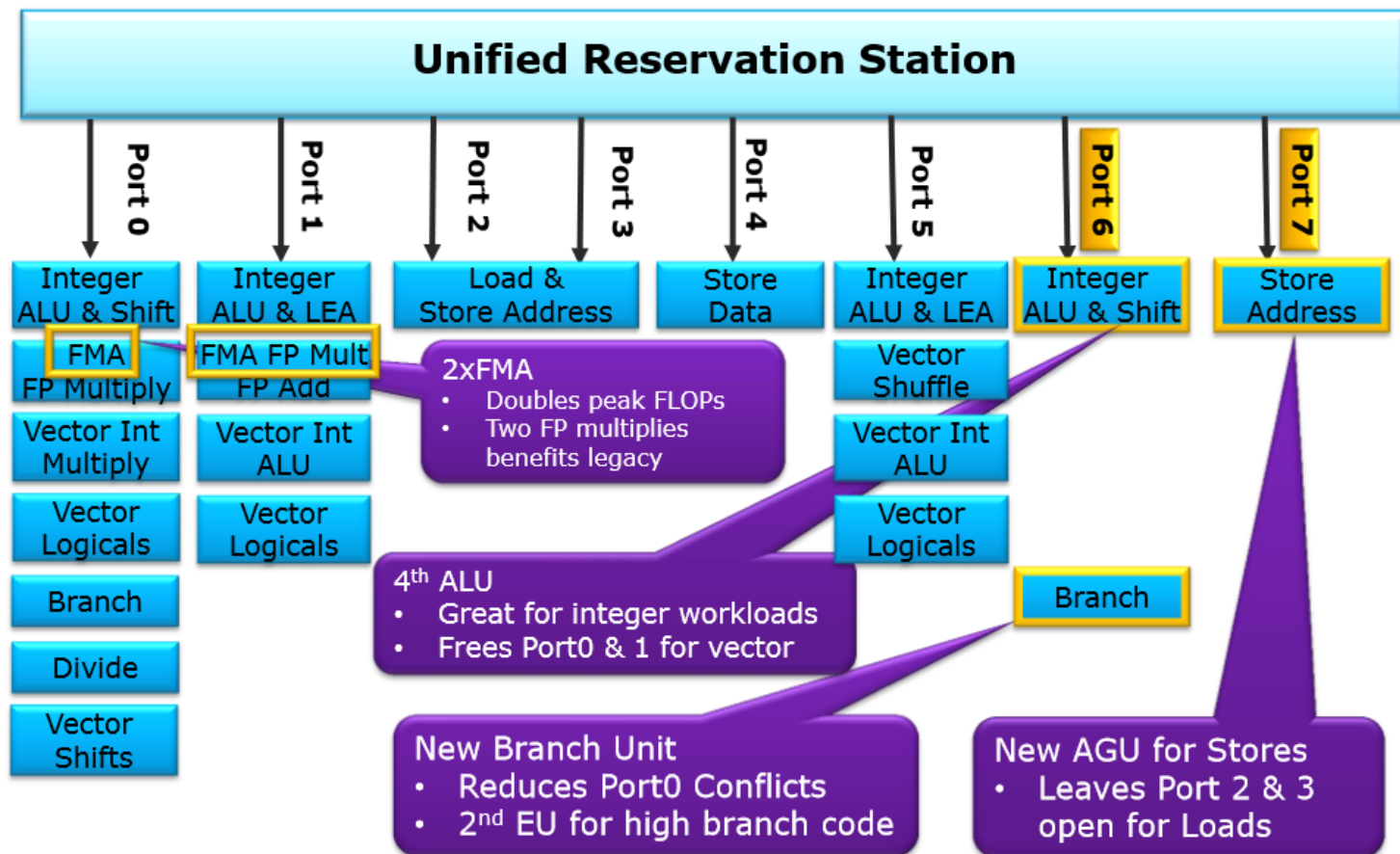
Benchmark	Mode	Score	Error	Units
testSequential	avgt	0.832	± 0.006	ns/op

~1 ns/op

Really???

Less than 1ns per operation?

Haswell Execution Unit Overview



Random walk per OS Page

-

**Average time in ns/op to sum all
longs in a 1GB array?**

Access Pattern Benchmark

Benchmark	Mode	Score	Error	Units
testSequential	avgt	0.832	± 0.006	ns/op
testRandomPage	avgt	2.703	± 0.025	ns/op

~3 ns/op

Data dependant walk per OS Page

-

**Average time in ns/op to sum all
longs in a 1GB array?**

Access Pattern Benchmark

Benchmark	Mode	Score	Error	Units
testSequential	avgt	0.832	± 0.006	ns/op
testRandomPage	avgt	2.703	± 0.025	ns/op
testDependentRandomPage	avgt	7.102	± 0.326	ns/op

~7 ns/op

Random heap walk

-

**Average time in ns/op to sum all
longs in a 1GB array?**

Access Pattern Benchmark

Benchmark	Mode	Score	Error	Units
testSequential	avgt	0.832	± 0.006	ns/op
testRandomPage	avgt	2.703	± 0.025	ns/op
testDependentRandomPage	avgt	7.102	± 0.326	ns/op
testRandomHeap	avgt	19.896	± 3.110	ns/op

~20 ns/op

Data dependant heap walk

-

**Average time in ns/op to sum all
longs in a 1GB array?**

Access Pattern Benchmark

Benchmark	Mode	Score	Error	Units
testSequential	avgt	0.832	± 0.006	ns/op
testRandomPage	avgt	2.703	± 0.025	ns/op
testDependentRandomPage	avgt	7.102	± 0.326	ns/op
testRandomHeap	avgt	19.896	± 3.110	ns/op
testDependentRandomHeap	avgt	89.516	± 4.573	ns/op

~90 ns/op

Then ADD **40+ ns/op**
for NUMA access on a server!!!!

**Data Dependent Loads
aka “Pointer Chasing”!!!**

Performance 101

Performance 101

1. **Memory is transported in Cachelines**

Performance 101

1. **Memory is transported in Cachelines**
2. **Memory is managed in OS Pages**

Performance 101

1. **Memory is transported in Cachelines**
2. **Memory is managed in OS Pages**
3. **Memory is pre-fetched on predictable access patterns**

Runtime Optimisation

Runtime JIT

1. Profile guided optimisations

Runtime JIT

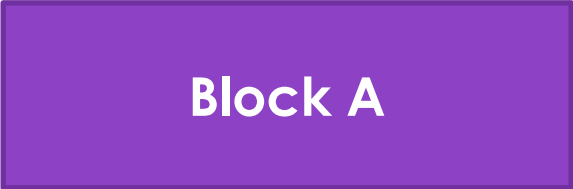
- 1. Profile guided optimisations**
- 2. Bets can be taken and later revoked**

Branches

```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```

Branches

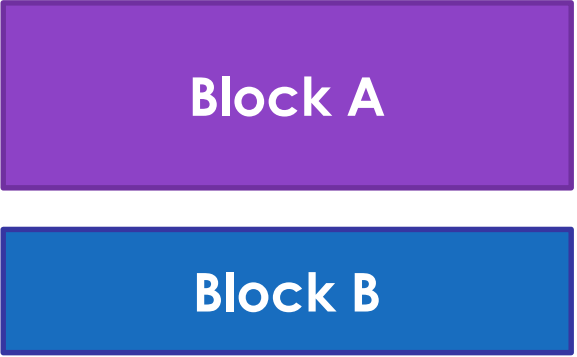
```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```



Block A

Branches

```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```



Block A

Block B

Branches

```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```

Block A

Block B

Block C

Branches

```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```

Block A

Block A

Block B

Block C

Block C

Branches

```
void foo()  
{  
    // code  
  
    if (condition)  
    {  
        // code  
    }  
  
    // code  
}
```



Subtle Branches

```
int result = (i > 7) ? a : b;
```

Subtle Branches

```
int result = (i > 7) ? a : b;
```

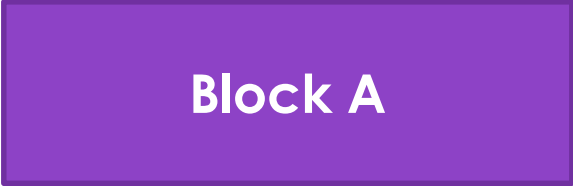
CMOV vs Branch Prediction?

Method/Function Inlining

```
void foo()  
{  
    // code  
  
    bar();  
  
    // code  
}
```

Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



Block A

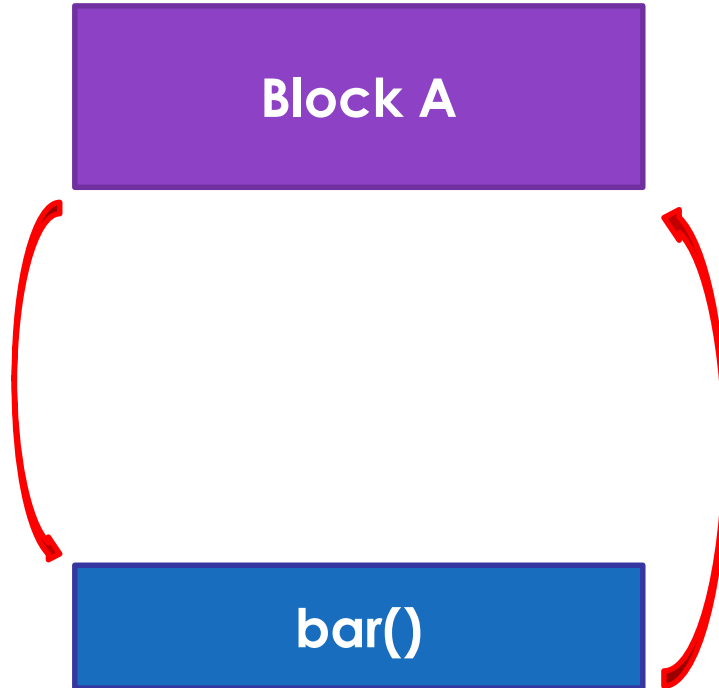
Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



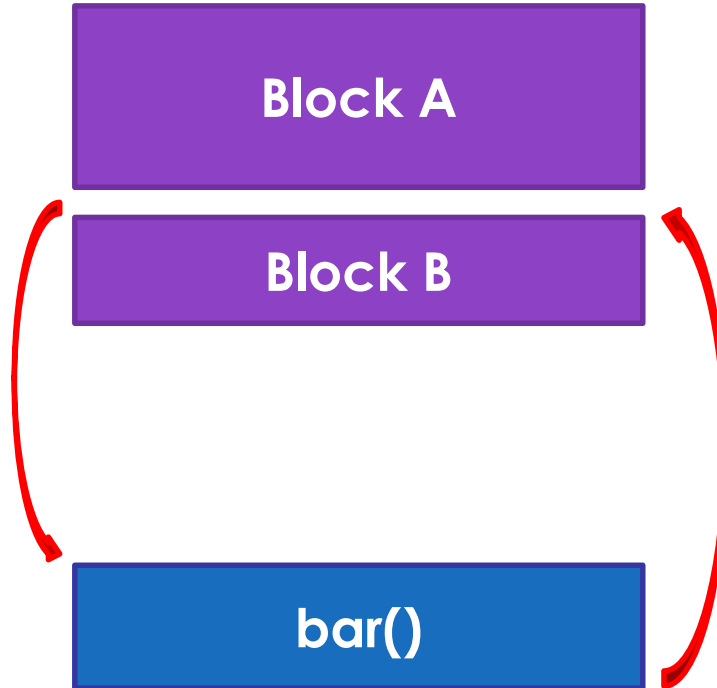
Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



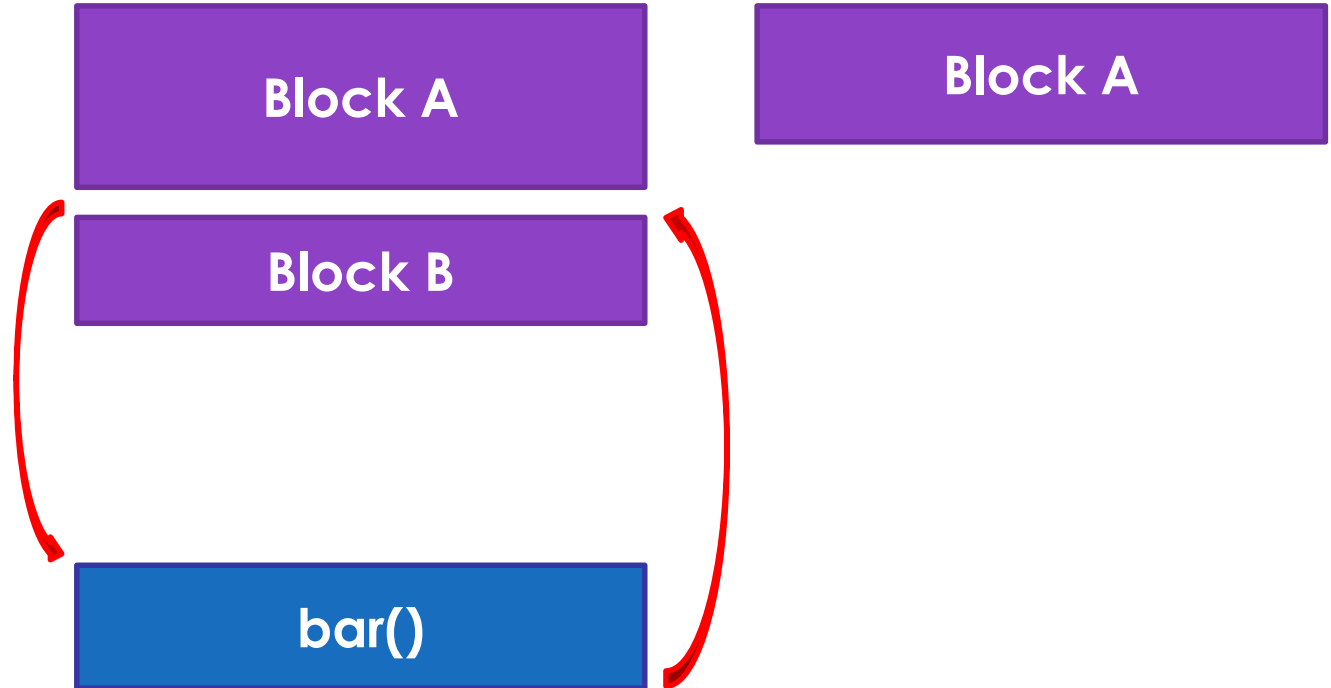
Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



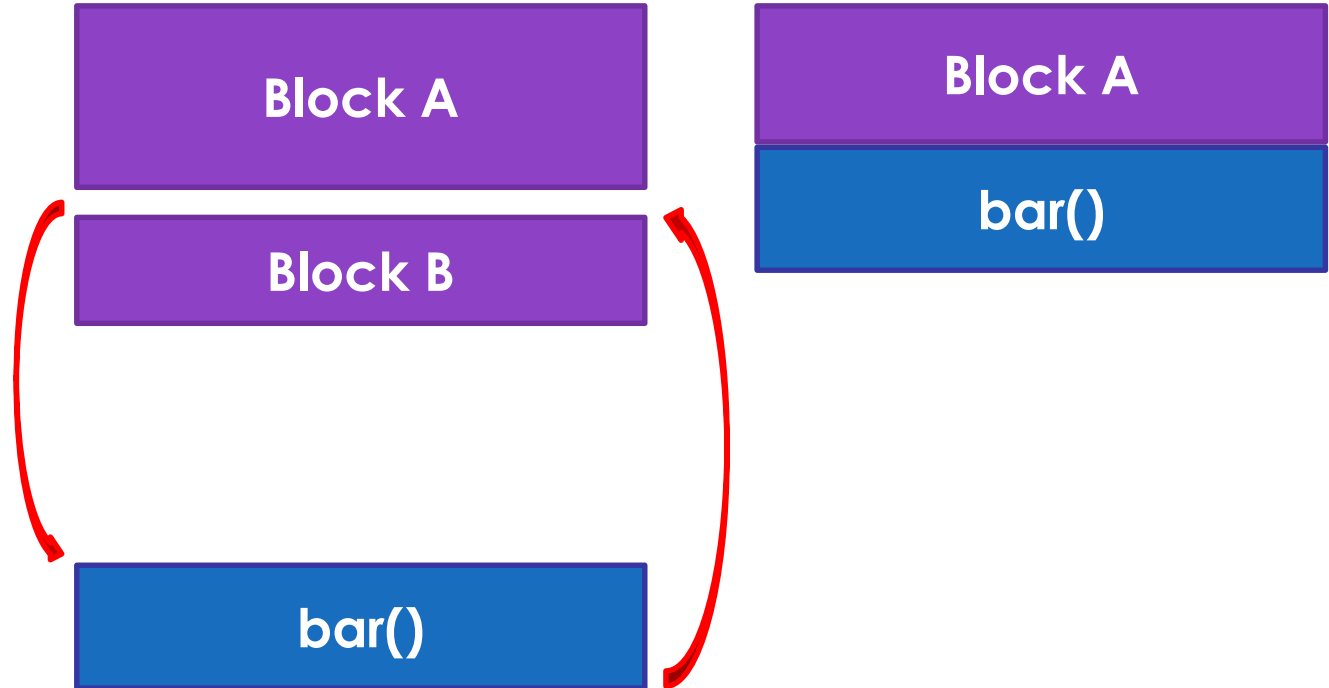
Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



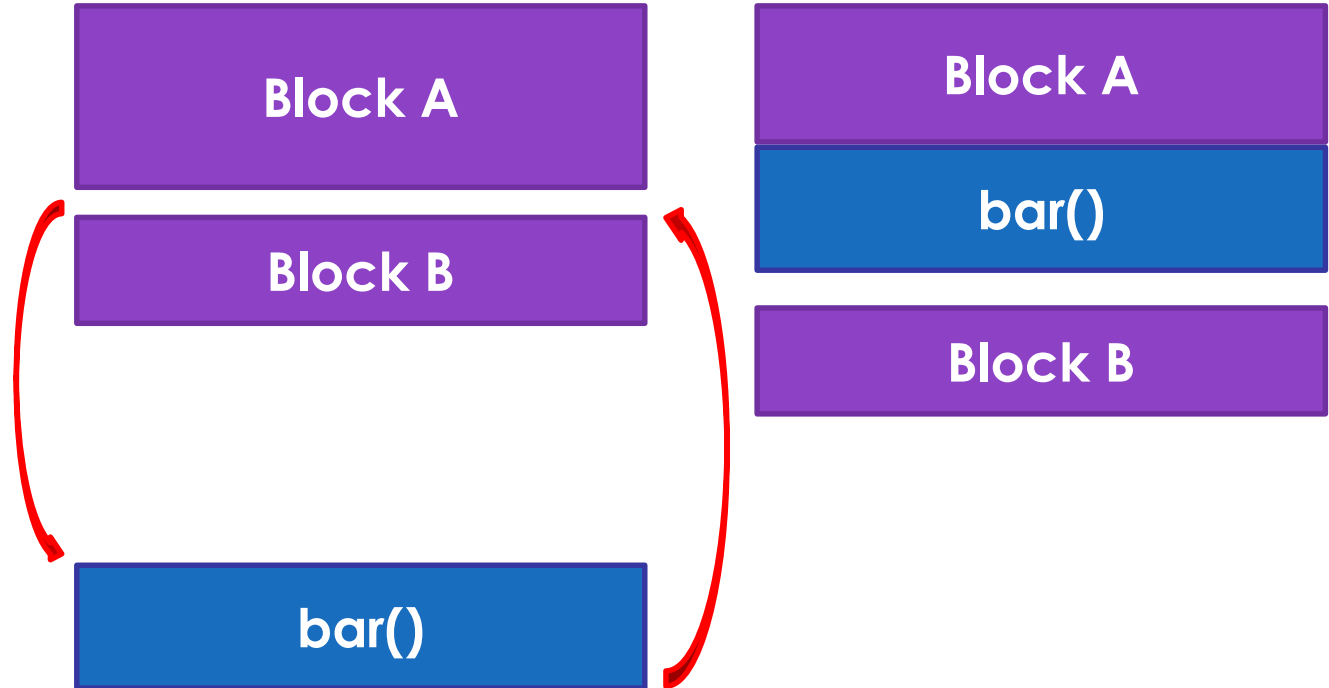
Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



Method/Function Inlining

```
void foo()  
{  
    // code  
    bar();  
    // code  
}
```



Method/Function Inlining

```
void foo()  
{  
    // code  
  
    bar();  
  
    // code  
}
```

**i-cache
& code bloat?**

Method/Function Inlining

“Inlining is THE optimisation.”

- Cliff Click

Bounds Checking

```
void foo(int[] array, int length)
{
    // code

    for (int i = 0; i < length; i++)
    {
        bar(array[i]);
    }

    // code
}
```

Bounds Checking

```
void foo(int[] array)
{
    // code

    for (int i = 0; i < array.length; i++)
    {
        bar(array[i]);
    }

    // code
}
```

Subtype Polymorphism

```
void draw(Shape[] shapes)
{
    for (int i = 0; i < shapes.length; i++)
    {
        shapes[i].draw();
    }
}
```

```
void bar(Shape shape)
{
    bar(shape.isVisible());
}
```

Subtype Polymorphism

```
void draw(Shape[] shapes)
{
    for (int i = 0; i < shapes.length; i++)
    {
        shapes[i].draw();
    }
}
```

```
void bar(Shape shape)
{
    bar(shape.isVisible());
}
```

**Class Hierarchy Analysis
& Inline Caching**

Runtime JIT

- 1. Profile guided optimisations**
- 2. Bets can be taken and later revoked**

Garbage Collection

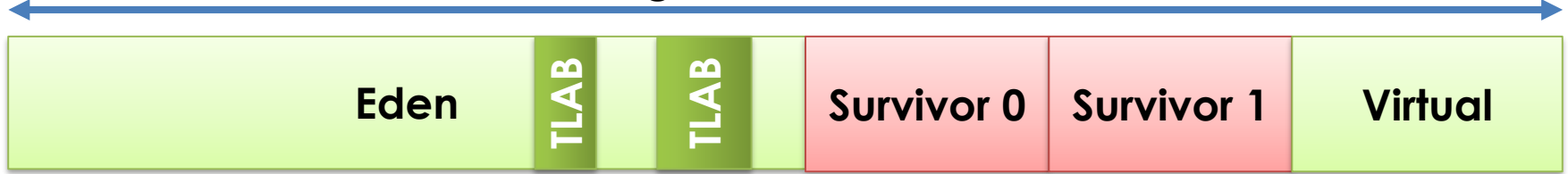
Generational Garbage Collection

“Only the good die young.”

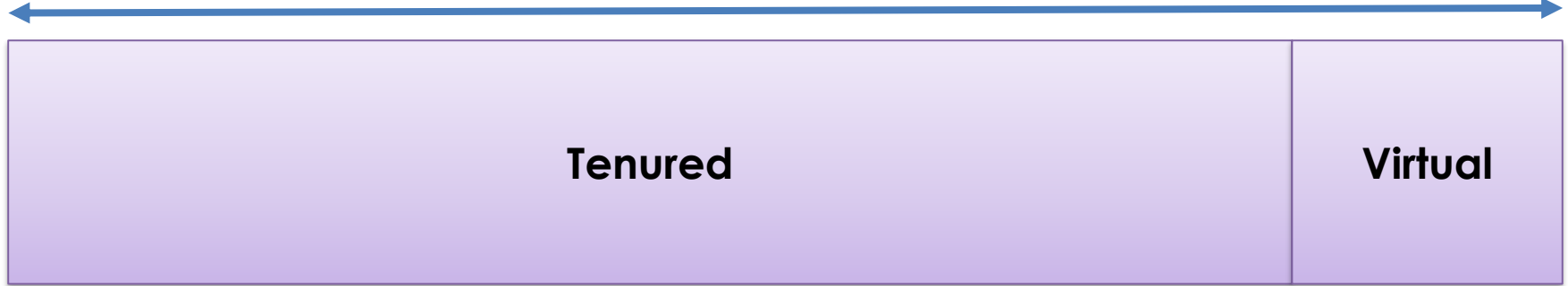
- Billy Joel

Generational Garbage Collection

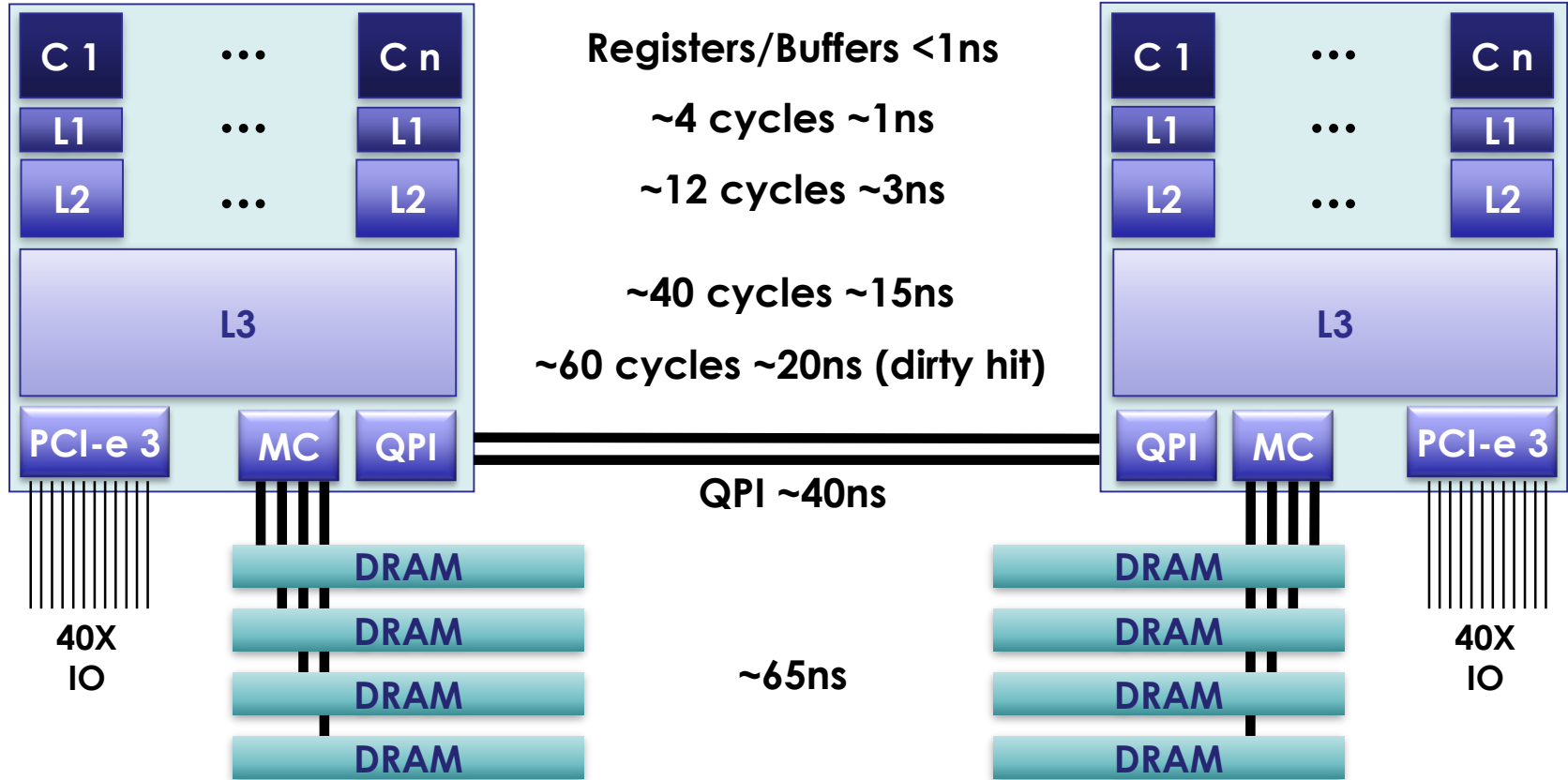
Young/New Generation



Old Generation



Modern Hardware (Intel Sandy Bridge EP)



Thread Local Allocation Buffers

Young/New Generation



Thread Local Allocation Buffers

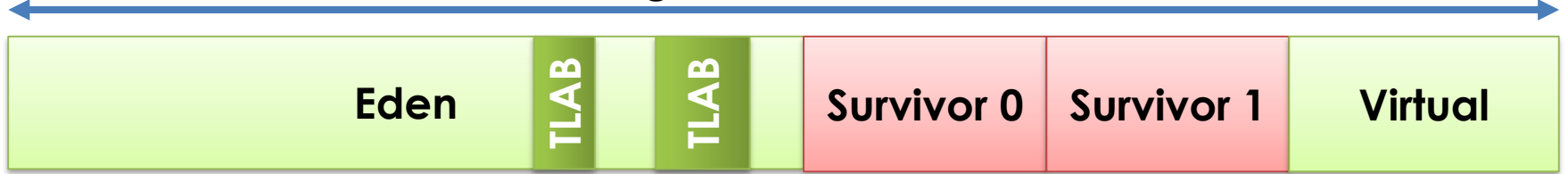
Young/New Generation



- Affords locality of reference
- Avoid false sharing
- Can have NUMA aware allocation

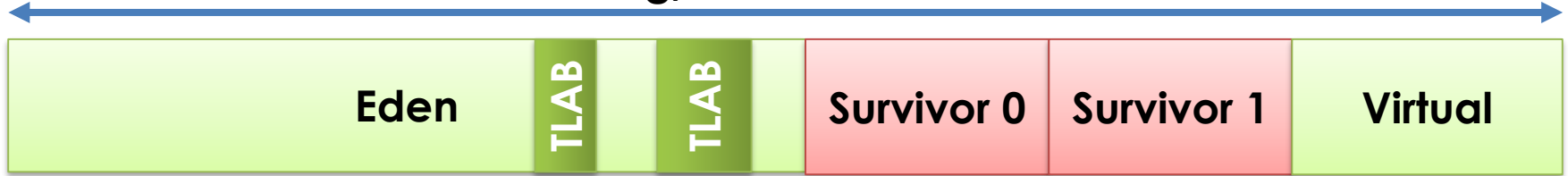
Object Survival

Young/New Generation



Object Survival

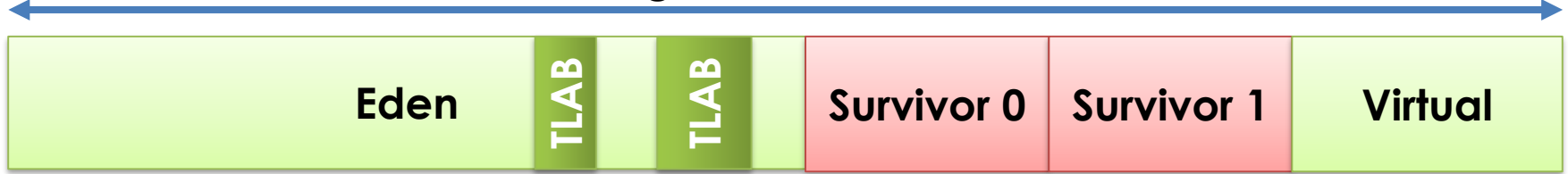
Young/New Generation



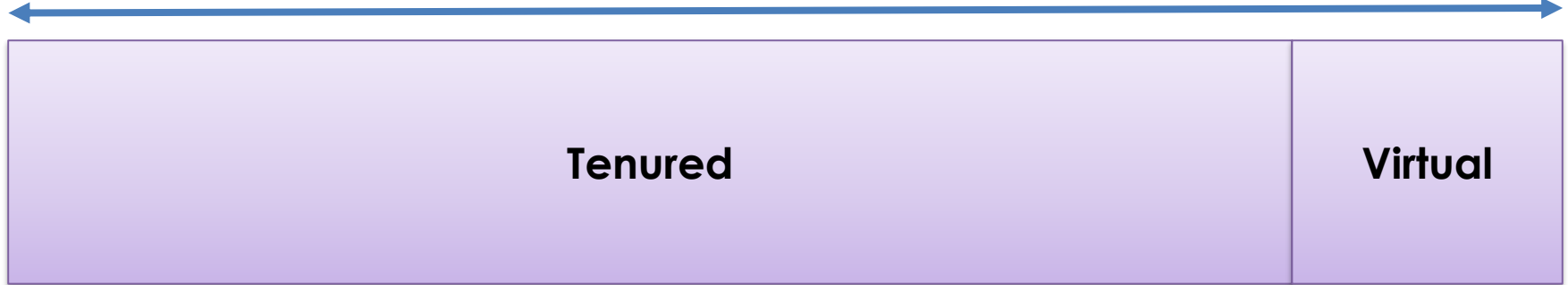
- Aging Policies
- Compacting Copy
- NUMA Interleave
- Fast Parallel Scavenging
- Only the survivors require work

Object Promotion

Young/New Generation

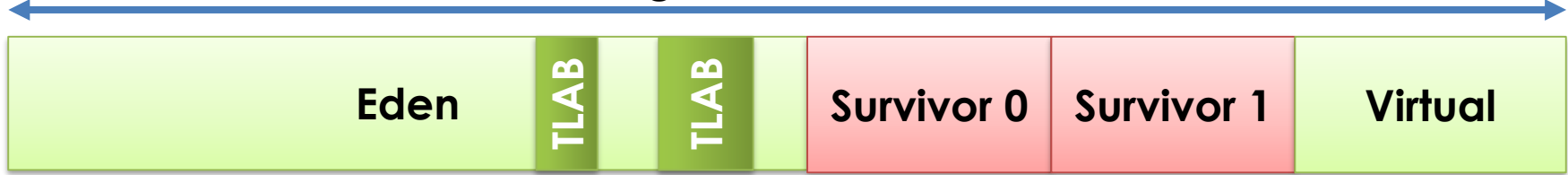


Old Generation

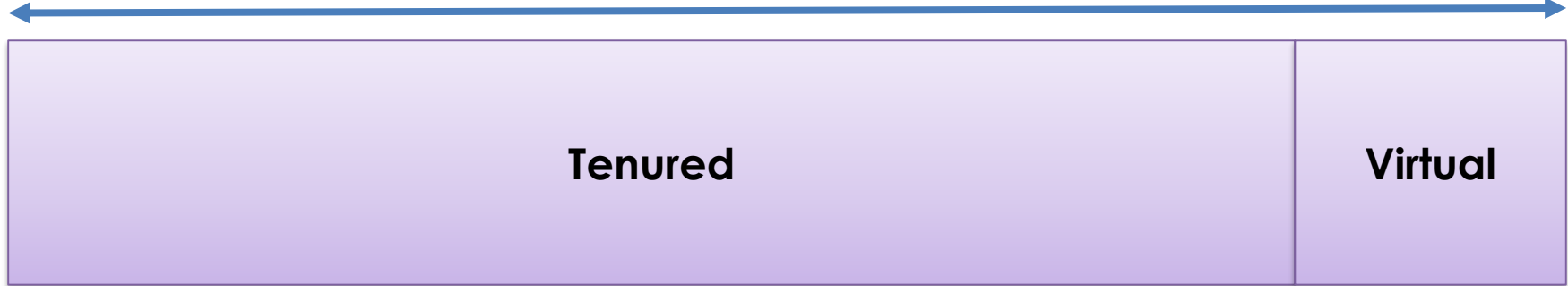


Object Promotion

Young/New Generation

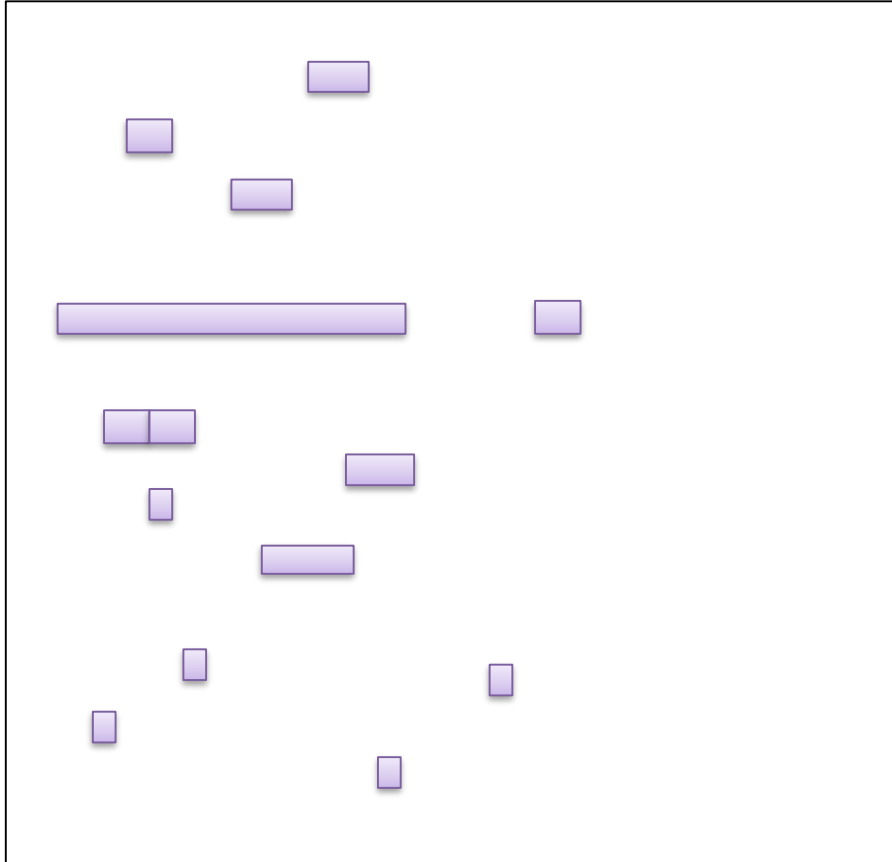


Old Generation



- Concurrent Collection
- String Deduplication

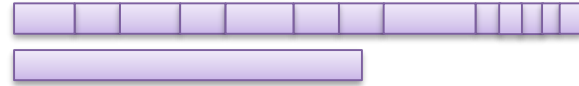
Compacting Collections



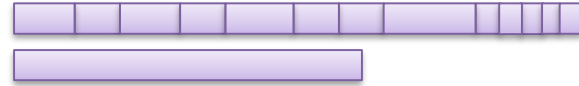
Compacting Collections – Depth first copy



Compacting Collections

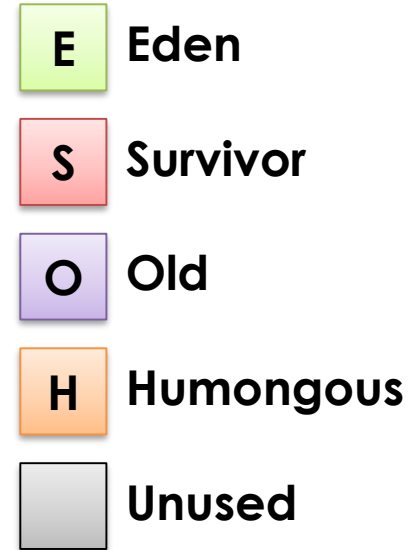
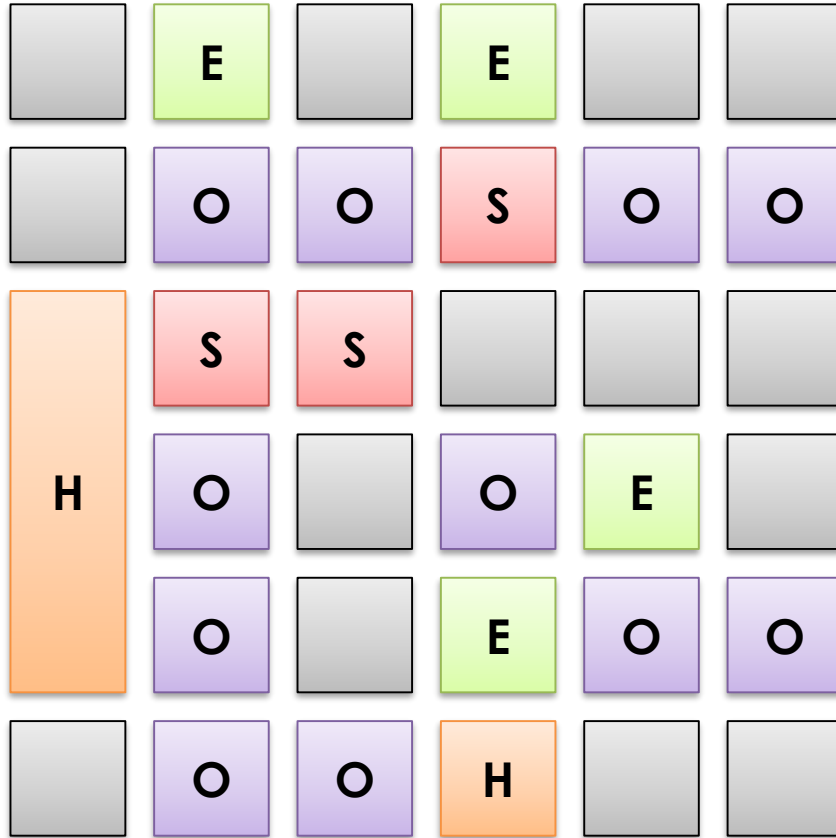


Compacting Collections



**OS Pages and
cache lines?**

G1 – Concurrent Compaction



Azul Zing C4

True Concurrent Compacting Collector

Where next for GC?

Object Inlining/Aggregation

GC vs Manual Memory Management

Not easy to pick clear winner...

GC vs Manual Memory Management

Not easy to pick clear winner...

Managed GC

- **GC Implementation**
- **Card Marking**
- **Read/Write Barriers**
- **Object Headers**
- **Background Overhead
in CPU and Memory**

GC vs Manual Memory Management

Not easy to pick clear winner...

Managed GC

- GC Implementation
- Card Marking
- Read/Write Barriers
- Object Headers
- Background Overhead in CPU and Memory

Native

- Malloc Implementation
- Arena/pool contention
- Bin Wastage
- Fragmentation
- Debugging Effort
- Inter-thread costs

Algorithms & Design

**What is most important to
performance?**

- **Avoiding cache misses**
- **Strength Reduction**
- **Avoiding duplicate work**
- **Amortising expensive operations**
- **Mechanical Sympathy**
- **Choice of Data Structures**
- **Choice of Algorithms**
- **API Design**
- **Overall Design**

**In a large codebase it is really
difficult to do everything well**

**It also takes some “*uncommon*”
disciplines such as:
profiling, telemetry, modelling...**

***“If I had more time, I would
have written a shorter letter.”***

- Blaise Pascal

The story of Aeron

**Aeron is an interesting lesson in
“time to performance”**

**Lots of others exists such at the
C# Roslyn compiler**

Time spent on

Mechanical Sympathy

vs

Debugging Pointers

???

Immutable Data & Concurrency

Functional Programming

In Closing ...

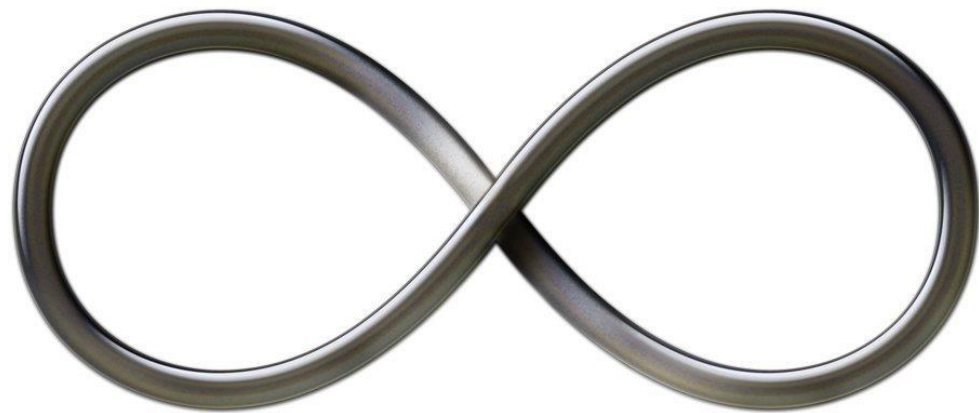
What does the future hold?

Remember
Assembly vs Compiled
Languages

**What about the issues of
footprint, startup time,
GC pauses, etc. ???**







Questions?

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

Twitter: @mjpt777

“Any intelligent fool can make things bigger, more complex, and more violent.

It takes a touch of genius, and a lot of courage, to move in the opposite direction.”

- Albert Einstein