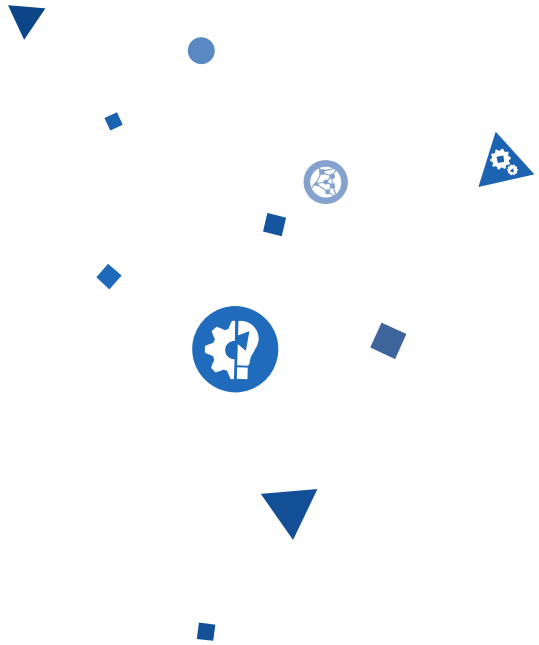


Runtime vs. compile time (JIT vs AOT) optimizations in Java and C++

Ionuţ Balosin

Java Software Architect

@ionutbalosin

Agenda

- Sequential sum of N elements array ($\sum_{i=1}^{N} array[i]$)
- Sequential sum of N integers ($\sum_{i=1}^{N} i$)
- Fields Layout
- Null Checks
- Lock Elision
- Virtual Calls
- Scalar Replacement
- Conclusions

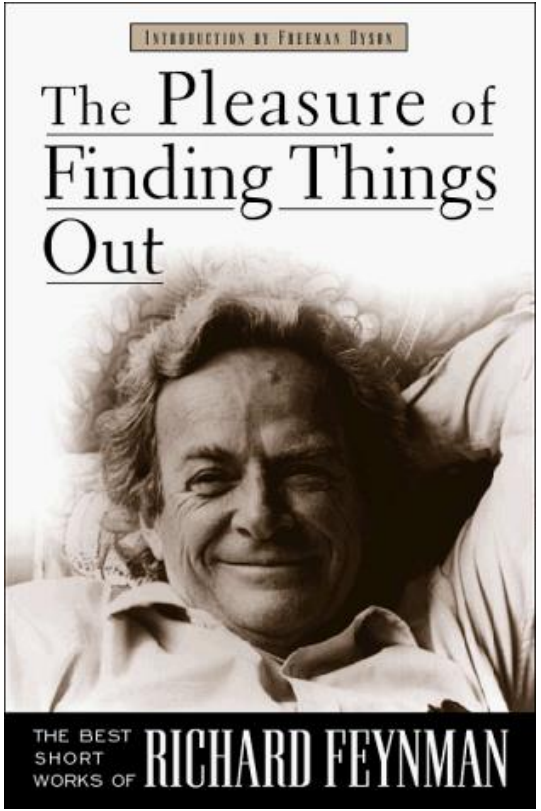
Disclaimer

```
0x00007f92409afe74: lock    add DWORD PTR [rsp],0x0
0x00007f92409afe79: cmp     QWORD PTR [r10+0x46],0x0
0x00007f92409afe81: jne     0x00007f92409afe9a
0x00007f92409afe83: mov     rax,0x0
0x00007f92409afe8d: lock    cmpxchg QWORD PTR [r10+0x16],r15
0x00007f92409afe93: jne     0x00007f92409afe9a
0x00007f92409afe95: or      rax,0x1
0x00007f92409afe98: jmp     0x00007f92409afeaa
0x00007f92409afe9a: test    rax,0x0
0x00007f92409afe9f: jne     0x00007f92409afeaa
0x00007f92409afea1: mov     rax,QWORD PTR [rax]
0x00007f92409afea4: cmp     QWORD PTR [rbp+0x0],r10
0x00007f92409afeaa: 
0x00007f92409afea
```

X86-64 Assembly Code

```
  4.988.081                                                   ( +- 19,16% )
  4.291.734                 #    0,86  insn per cycle         ( +-  4,68% )
  2.533.476                                                   ( +- 18,41% )
  1.397.434      cache-misses             #   55,159 % of all cache refs      ( +-  1,73% )
    429.334      bus-cycles                                   ( +-  2,17% )
 17.079.297      L1-dcache-loads                              ( +-  8,63% )  (90,89%)
 17.552.174      L1-dcache-load-misses    #  102,77% of all L1-dcache hits    ( +-  3,59% )  (60,07%)
    425.336      L1-dcache-stores                             ( +- 12,01% )  (46,94%)
 35.469.510      dTLB-loads                                   ( +-  3,36% )  (32,65%)
      1.980      dTLB-load-misses         #    0,01% of all dTLB cache hits   ( +- 37,20% )  (17,67%)
```

Talk Motivation



Talk Motivation

The “Pleasure of Findings Things Out”

Learning based on evidence driven by experiments

Key Points

What optimizations are generated at runtime by JIT C2 in comparison to compile time optimizations triggered by LLVM clang?

How powerful are runtime (i.e. JIT C2) vs. ahead of time (i.e. LLVM clang) optimizations?

This presentation is **not a battle**, I **do not** try to **establish a winner**, but just to study different optimization approaches!

Why it might be a matter of interest?

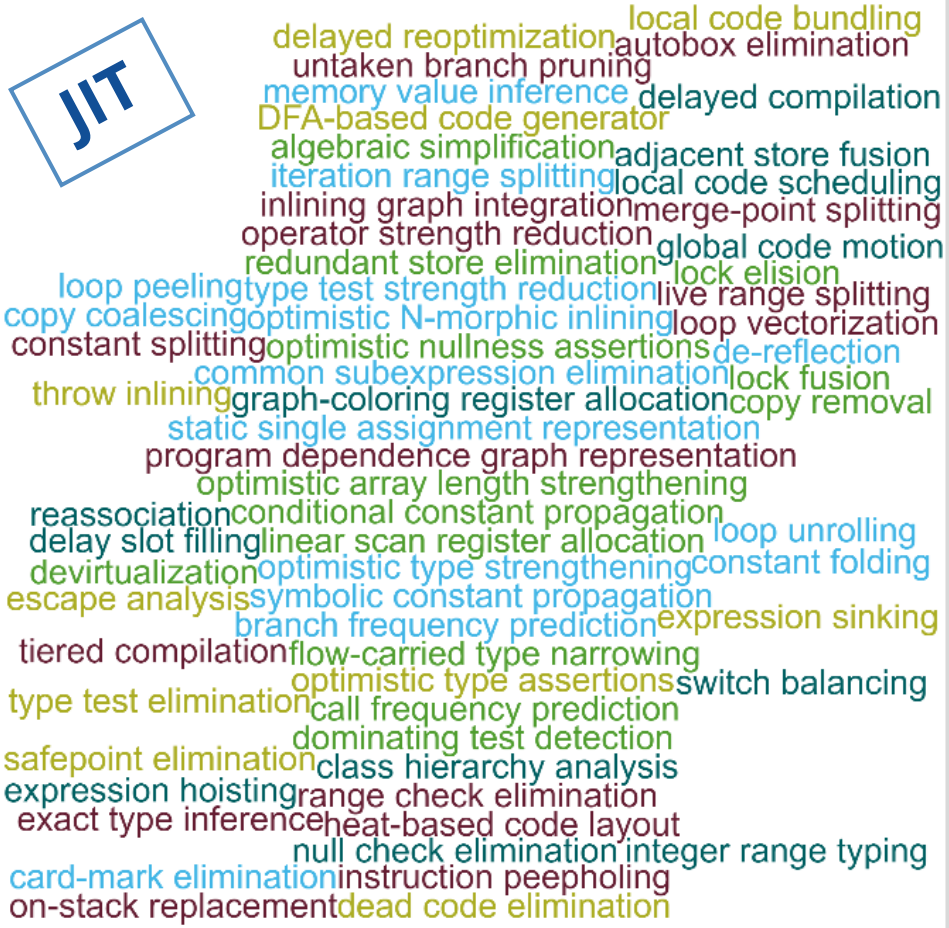
Performance

LLVM Clang is one of the best AOT Compilers

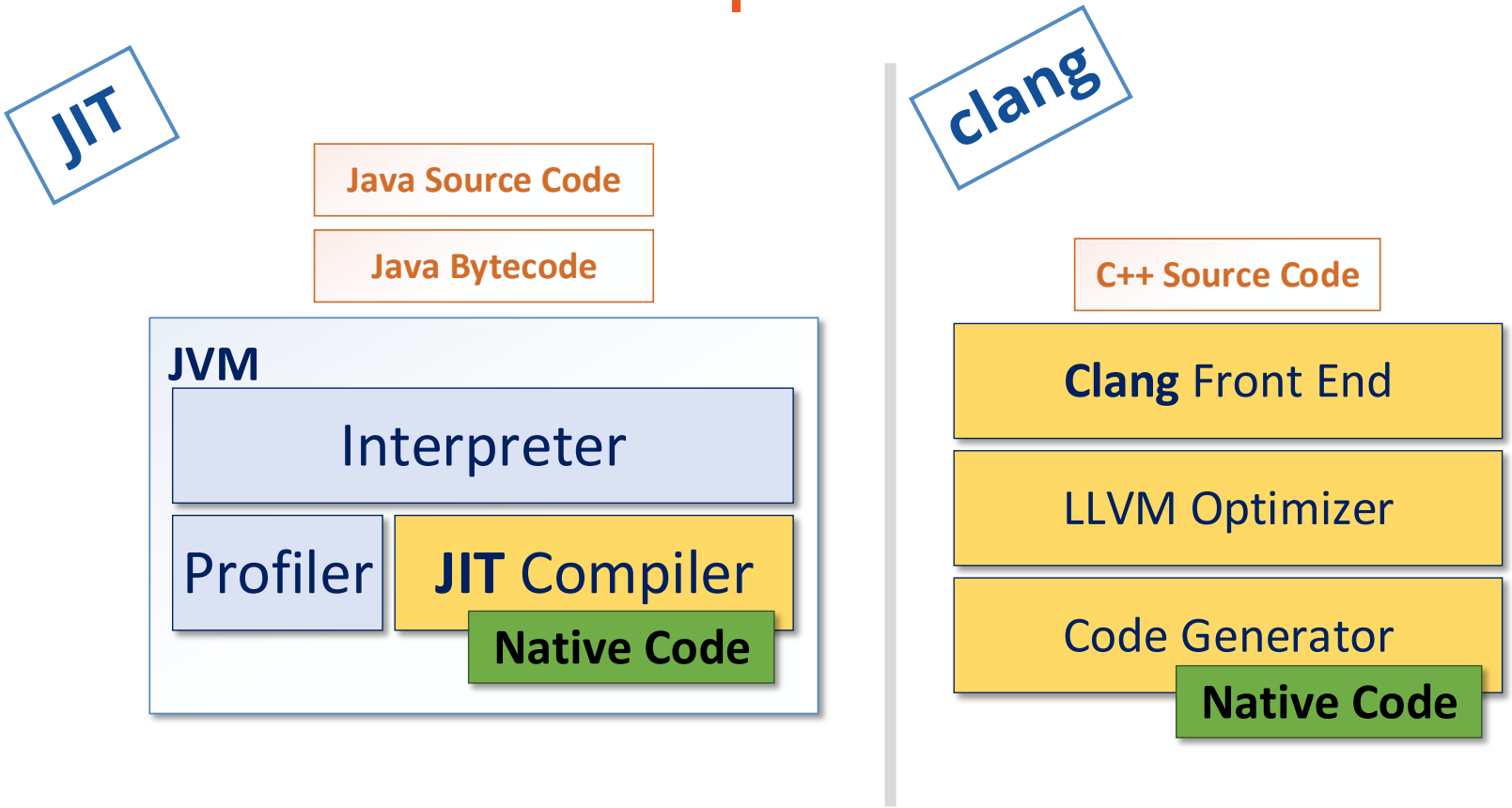
Java is moving towards AOT (JEP-295)

JIT C2 is still the most popular option for Java

Worlds of Optimizations



Compilation Process



Tests Configuration

Hardware

- Intel i7-6700HQ Skylake
- 16GB DDR4 2133 MHz

Software

- Ubuntu 16.04.2
- JDK 9.0.1 x64
- Java Measurement Harness
- JITWatch
- LLVM clang 5.0.0 x86-84
- C++ Google Benchmark
- gcc.godbolt.org
- perf

Tests Pattern

```java
@Benchmark
public long hotMethod() {
    long sum = 0;
    for (int i = 0; i < array.length; i++) {
        sum += array[i];
    }
```

Java Source Code

```
add ebx,DWORD PTR [r10+r13*4+0x10]
movsxd r9,r13d
add ebx,DWORD PTR [r10+r9*4+0x14]
add ebx,DWORD PTR [r10+r9*4+0x18]
add ebx,DWORD PTR [r10+r9*4+0x1c]
add ebx,DWORD PTR [r10+r9*4+0x20]
add ebx,DWORD PTR [r10+r9*4+0x24]
```

JIT C2 generated code

```
movdqu xmm2, xmmword ptr [rc
paddd xmm2, xmm0
movdqu xmm0, xmmword ptr [rc
paddd xmm0, xmm1
movdqu xmm0, xmmword ptr [rc
paddd xmm0, xmm4
```

LLVM clang generated code

JIT C2 (ns/op)	LLVM Clang (ns/op)
2.8	190

Benchmark Comparison

LLVM clang cannot optimize the lock synchronized block.
Runtime performance is dependent

JIT C2 is able to remove locks which performance.
Runtime performance is not depend

Conclusions

Methodology and Pitfalls

Comparing LLVM clang vs. JIT C2 optimizations using different programming languages is difficult:

- it is not always  to  (e.g. not the same source code)
- language specifics (e.g. Java, C++)
- benchmarks specifics (e.g. measurements reported in nanos w/o decimal precision)
- run on single machine and report
- run using single Java compiler (e.g. JIT C2) and report
- “fast enough is fast enough” – Cliff Click

Methodology and Pitfalls

Comparing LLVM clang vs. JIT C2 optimizations using different programming languages is difficult:

- it is not always to (e.g. not the same source code)
- language specifics (e.g. Java, C++)
- benchmarks specifics (e.g. measurements reported in nanos w/o decimal precision)
- run using single Java compiler (e.g. JIT C2)
- “fast enough is fast enough” – Cliff Click

The aim is to **reveal** few **under the hood optimizations** focusing less on timing measurements!

Sequential sum of N elements array

$$\left(\sum_{i=1}^{N} array[i]\right)$$

Sequential sum of N elements array

Source code

```
private int[] array;


@Benchmark
public long hotMethod() {
    long sum = 0;

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

    return sum;
}
```

Sequential sum of N elements array

JIT C2

```
                mov r13d,DWORD PTR [rdx]            ; r13d - local index
                mov r8d,r13d
                inc r8d                             ; r8 <- 1

+ L0:           cmp r13d,r11d                       ; r11d - is the array length
|               jae L5
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r8d
+               jl L0

+ L1:           add ebx,DWORD PTR [r10+r13*4+0x10]
|               movsxd r9,r13d
|               add ebx,DWORD PTR [r10+r9*4+0x14]
|               add ebx,DWORD PTR [r10+r9*4+0x18]
|               add ebx,DWORD PTR [r10+r9*4+0x1c]
|               add ebx,DWORD PTR [r10+r9*4+0x20]
|               add ebx,DWORD PTR [r10+r9*4+0x24]
|               add ebx,DWORD PTR [r10+r9*4+0x28]
|               add ebx,DWORD PTR [r10+r9*4+0x2c]
|               add r13d,0x8
|               cmp r13d,r8d
+               jl L1

+ L3:           cmp r13d,r11d
|               jae L6     ;*iaload
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r11d
+               jl L3

                mov eax,ebx
```

Sequential sum of N elements array

JIT C2

scalar pre loop
- 1 integer add per cycle -

```
                mov r13d,DWORD PTR [rdx]            ; r13d - local index
                mov r8d,r13
                inc r8d                             ; r8 <- 1

+ L0:           cmp r13d,r11d                       ; r11d - is the array length
|               jae L5
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r8d
+               jl L0

+ L1:           add ebx,DWORD PTR [r10+r13*4+0x10]
|               movsxd r9,r13d
|               add ebx,DWORD PTR [r10+r9*4+0x14]
|               add ebx,DWORD PTR [r10+r9*4+0x18]
|               add ebx,DWORD PTR [r10+r9*4+0x1c]
|               add ebx,DWORD PTR [r10+r9*4+0x20]
|               add ebx,DWORD PTR [r10+r9*4+0x24]
|               add ebx,DWORD PTR [r10+r9*4+0x28]
|               add ebx,DWORD PTR [r10+r9*4+0x2c]
|               add r13d,0x8
|               cmp r13d,r8d
+               jl L1

+ L3:           cmp r13d,r11d
|               jae L6     ;*iaload
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r11d
+               jl L3

                mov eax,ebx
```

Sequential sum of N elements array

JIT C2

scalar pre loop
- 1 integer add per cycle -

scalar main loop
loop unrolling
- 8 integer adds per cycle -

```
                mov r13d,DWORD PTR [rdx]            ; r13d - local index
                mov r8d,r13
                inc r8d                             ; r8 <- 1

+ L0:           cmp r13d,r11d                       ; r11d - is the array length
|               jae L5
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r8d
+               jl L0

+ L1:           add ebx,DWORD PTR [r10+r13*4+0x10]
|               movsxd r9,r13d
|               add ebx,DWORD PTR [r10+r9*4+0x14]
|               add ebx,DWORD PTR [r10+r9*4+0x18]
|               add ebx,DWORD PTR [r10+r9*4+0x1c]
|               add ebx,DWORD PTR [r10+r9*4+0x20]
|               add ebx,DWORD PTR [r10+r9*4+0x24]
|               add ebx,DWORD PTR [r10+r9*4+0x28]
|               add ebx,DWORD PTR [r10+r9*4+0x2c]
|               add r13d,0x8
|               cmp r13d,r8d
+               jl L1

+ L3:           cmp r13d,r11d
|               jae L6     ;*iaload
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r11d
+               jl L3

                mov eax,ebx
```

Sequential sum of N elements array

JIT C2

scalar pre loop
- 1 integer add per cycle -

scalar main loop
loop unrolling
- 8 integer adds per cycle -

scalar post loop
- 1 integer add per cycle -

```
                mov r13d,DWORD PTR [rdx]            ; r13d - local index
                mov r8d,r13
                inc r8d                             ; r8 <- 1

+ L0:           cmp r13d,r11d                       ; r11d - is the array length
|               jae L5
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r8d
+               jl L0

+ L1:           add ebx,DWORD PTR [r10+r13*4+0x10]
|               movsxd r9,r13d
|               add ebx,DWORD PTR [r10+r9*4+0x14]
|               add ebx,DWORD PTR [r10+r9*4+0x18]
|               add ebx,DWORD PTR [r10+r9*4+0x1c]
|               add ebx,DWORD PTR [r10+r9*4+0x20]
|               add ebx,DWORD PTR [r10+r9*4+0x24]
|               add ebx,DWORD PTR [r10+r9*4+0x28]
|               add ebx,DWORD PTR [r10+r9*4+0x2c]
|               add r13d,0x8
|               cmp r13d,r8d
+               jl L1

+ L3:           cmp r13d,r11d
|               jae L6     ;*iaload
|               add ebx,DWORD PTR [r10+r13*4+0x10]
|               inc r13d
|               cmp r13d,r11d
+               jl L3

                mov eax,ebx
```

Sequential sum of N elements array

LLVM clang

```
+ L0:   movdqu xmm2, xmmword ptr [rcx + 4*rdi]
|       paddd xmm2, xmm0
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 16]
|       paddd xmm0, xmm1
|       ...
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 96]
|       paddd xmm0, xmm4
|       movdqu xmm1, xmmword ptr [rcx + 4*rdi + 112]
|       paddd xmm1, xmm2
|       add rdi, 32
|       add rdx, 4
+       jne L0

+ L1:   movdqu xmm2, xmmword ptr [rdx - 16]
|       paddd xmm0, xmm2
|       movdqu xmm2, xmmword ptr [rdx]
|       paddd xmm1, xmm2
|       add rdx, 32
|       inc rax
+       jne L1

+ L2:   add eax, dword ptr [rcx + 4*rsi]
|       inc rsi
|       cmp rsi, r9
+       jb L2
```

Sequential sum of N elements array

LLVM clang

vectorized main loop
loop unrolling
- 8x4 adds per cycle -

```
+ L0:   movdqu xmm2, xmmword ptr [rcx + 4*rdi]
|       paddd xmm2, xmm0
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 16]
|       paddd xmm0, xmm1
|       ...
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 96]
|       paddd xmm0, xmm4
|       movdqu xmm1, xmmword ptr [rcx + 4*rdi + 112]
|       paddd xmm1, xmm2
|       add rdi, 32
|       add rdx, 4
+       jne L0

+ L1:   movdqu xmm2, xmmword ptr [rdx - 16]
|       paddd xmm0, xmm2
|       movdqu xmm2, xmmword ptr [rdx]
|       paddd xmm1, xmm2
|       add rdx, 32
|       inc rax
+       jne L1

+ L2:   add eax, dword ptr [rcx + 4*rsi]
|       inc rsi
|       cmp rsi, r9
+       jb L2
```

[note] 1 add - 1 array integer add

Sequential sum of N elements array

LLVM clang

vectorized main loop
loop unrolling
- 8x4 adds per cycle -

```
+ L0:   movdqu xmm2, xmmword ptr [rcx + 4*rdi]
|       paddd xmm2, xmm0
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 16]
|       paddd xmm0, xmm1
|       ...
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 96]
|       paddd xmm0, xmm4
|       movdqu xmm1, xmmword ptr [rcx + 4*rdi + 112]
|       paddd xmm1, xmm2
|       add rdi, 32
|       add rdx, 4
+       jne L0
```

vectorized post loop
loop unrolling
- 2x4 adds per cycle -

```
+ L1:   movdqu xmm2, xmmword ptr [rdx - 16]
|       paddd xmm0, xmm2
|       movdqu xmm2, xmmword ptr [rdx]
|       paddd xmm1, xmm2
|       add rdx, 32
|       inc rax
+       jne L1
```

```
+ L2:   add eax, dword ptr [rcx + 4*rsi]
|       inc rsi
|       cmp rsi, r9
+       jb L2
```

[note] 1 add - 1 array integer add

Sequential sum of N elements array

LLVM clang

vectorized main loop
loop unrolling
- 8x4 adds per cycle -

```
+ L0:   movdqu xmm2, xmmword ptr [rcx + 4*rdi]
|       paddd xmm2, xmm0
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 16]
|       paddd xmm0, xmm1
|       ...
|       movdqu xmm0, xmmword ptr [rcx + 4*rdi + 96]
|       paddd xmm0, xmm4
|       movdqu xmm1, xmmword ptr [rcx + 4*rdi + 112]
|       paddd xmm1, xmm2
|       add rdi, 32
|       add rdx, 4
+       jne L0
```

vectorized post loop
loop unrolling
- 2x4 adds per cycle -

```
+ L1:   movdqu xmm2, xmmword ptr [rdx - 16]
|       paddd xmm0, xmm2
|       movdqu xmm2, xmmword ptr [rdx]
|       paddd xmm1, xmm2
|       add rdx, 32
|       inc rax
+       jne L1
```

scalar post loop
- 1 add per cycle -

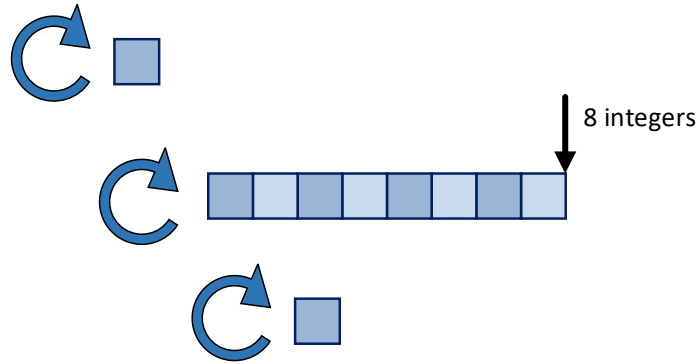
```
+ L2:   add eax, dword ptr [rcx + 4*rsi]
|       inc rsi
|       cmp rsi, r9
+       jb L2
```

[note] 1 add - 1 array integer add

Sequential sum of N elements array

JIT C2

- scalar pre loop
- scalar main loop
- scalar post loop



[note] pattern valid for this case

32 bit

128 bit

Sequential sum of N elements array

JIT C2

scalar pre loop

scalar main loop

scalar post loop

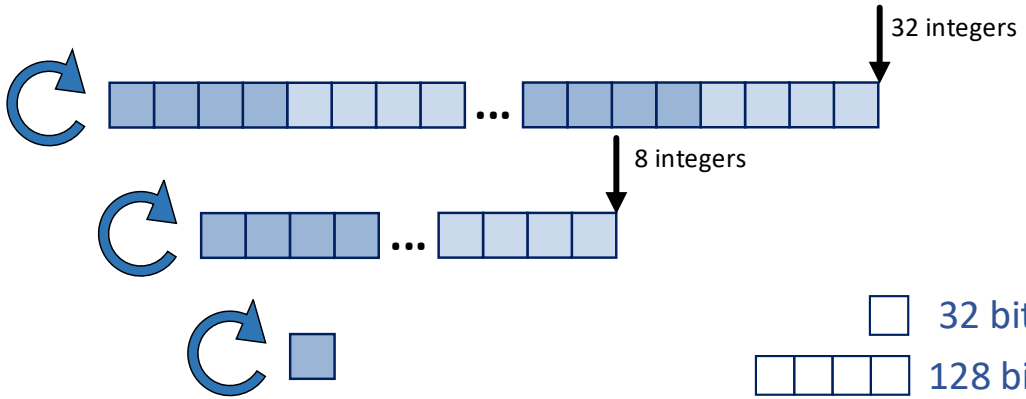
[note] pattern valid for this case

LLVM clang

vectorized main loop

vectorized post loop

scalar post loop



Sequential sum of N elements array

Benchmark comparison

Array Size	JIT C2 us/op	LLVM Clang us/op
10×10^3	3.2	0.8
100×10^3	30.1	9.6
$1\,000 \times 10^3$	328.9	98.6

lower is better

[note] 1 op – 1 method call

Sequential sum of N elements array

Conclusions

LLVM clang uses vectorization for:

- main loop - 5 XMM SSE registers on 128 bits → 32 integer adds per loop
- post loop - 3 XMM SSE registers on 128 bits → 8 integer adds per loop

LLVM clang does the remaining post loop without unrolling and without vectorization, just adding one by one

Optimizations done by JIT C2 (e.g. loop peeling, loop unrolling) without vectorization support could not outperform LLVM clang

Sequential sum of N elements array

How LLVM clang optimizes the loop if array size is constant (e.g. 128)?

Sequential sum of N elements array

Source code

```
@Benchmark
public long hotMethod() {
    long sum = 0;

    for (int i = 0; i < 128; i++) {
        sum += array[i];
    }

    return sum;
}
```

Sequential sum of N elements array

LLVM clang

```
mov rax, qword ptr [rdi]
movdqu xmm0, xmmword ptr [rax]
movdqu xmm1, xmmword ptr [rax + 16]
movdqu xmm2, xmmword ptr [rax + 32]
paddd xmm2, xmm0
movdqu xmm0, xmmword ptr [rax + 48]
paddd xmm0, xmm1
...
movdqu xmm2, xmmword ptr [rax + 448]
paddd xmm2, xmm3
movdqu xmm3, xmmword ptr [rax + 464]
paddd xmm3, xmm4
movdqu xmm4, xmmword ptr [rax + 480]
paddd xmm4, xmm2
paddd xmm4, xmm0
movdqu xmm0, xmmword ptr [rax + 496]
paddd xmm0, xmm3
paddd xmm0, xmm1
paddd xmm0, xmm4
...
movd eax, xmm0
```

Sequential sum of N elements array

LLVM clang

vectorized sum
no loop
- 128 integer adds -

```
mov rax, qword ptr [rdi]
movdqu xmm0, xmmword ptr [rax]
movdqu xmm1, xmmword ptr [rax + 16]
movdqu xmm2, xmmword ptr [rax + 32]
paddd xmm2, xmm0
movdqu xmm0, xmmword ptr [rax + 48]
paddd xmm0, xmm1
...
movdqu xmm2, xmmword ptr [rax + 448]
paddd xmm2, xmm3
movdqu xmm3, xmmword ptr [rax + 464]
paddd xmm3, xmm4
movdqu xmm4, xmmword ptr [rax + 480]
paddd xmm4, xmm2
paddd xmm4, xmm0
movdqu xmm0, xmmword ptr [rax + 496]
paddd xmm0, xmm3
paddd xmm0, xmm1
paddd xmm0, xmm4
...
movd eax, xmm0
```

Sequential sum of N elements array

LLVM clang

```
mov rax, qword ptr [rdi]
movdqu xmm0, xmmword ptr [rax]
movdqu xmm1, xmmword ptr [rax + 16]
movdqu xmm2, xmmword ptr [rax + 32]
paddd xmm2, xmm0
movdqu xmm0, xmmword ptr [rax + 48]
paddd xmm0, xmm1
...
movdqu xmm2, xmmword ptr [rax + 448]
movdqu xmm4, xmmword ptr [rax + 480]
paddd xmm4, xmm2
paddd xmm4, xmm0
movdqu xmm0, xmmword ptr [rax + 496]
paddd xmm0, xmm3
paddd xmm0, xmm1
paddd xmm0, xmm4
...
movd eax, xmm0
```

LLVM clang removes the number of jumps in spite of increasing the iCache size!

JDK / JDK-8188313

C2: Consider enabling auto-vectorization for simple reductions

Type:	Enhancement	Status:	OPEN
Priority:	P3	Resolution:	Unresolved
Affects Version/s:	9	Fix Version/s:	11
Component/s:	hotspot		
Labels:	c2 c2-loopopts community-candidate performance		
Subcomponent:	compiler		

Description

Reconsider the decision to disable autovectorization for simple reductions made by ~~JDK-8078563~~.

Richard Startin reports [1] that artificially complicating reduction operation (to make it eligible for auto-vectorization) produces a better result than original code (sum of elements).

[1] http://richardstartin.uk/tricking-java-into-adding-up-arrays-faster/

But … which are the cases when JIT does loop vectorization?

Sum(array[], array[])

```
private int[] a, int[] b, int[] c;
private int len;

@Setup(Level.Trial)
public void setUp() {
    this.len = 100_000_000;
    this.a = new int[len];
    this.b = new int[len];
    this.c = new int[len];
    for (int i = 0; i < len; i++) {
        a[i] = i + 1;
        b[i] = i + 1;
    }
}

@Benchmark
public int[] hotMethod() {
    for (int i = 0; i < len; i++) {
        c[i] = a[i] + b[i];
    }
    return c;
}
```

Sum(array[], array[])

JIT C2

vectorized main loop
loop unrolling
- 4x8 adds per cycle -

```
+   L0:   vmovdqu ymm0,YMMWORD PTR [rsi+rdi*4+0x10]
|         vpaddd  ymm0,ymm0,YMMWORD PTR [rdx+rdi*4+0x10]
|         vmovdqu YMMWORD PTR [r9+rdi*4+0x10],ymm0
|         movsxd  r8,edi
|         vmovdqu ymm0,YMMWORD PTR [rsi+r8*4+0x30]
|         vpaddd  ymm0,ymm0,YMMWORD PTR [rdx+r8*4+0x30]
|         vmovdqu YMMWORD PTR [r9+r8*4+0x30],ymm0
|         vmovdqu ymm0,YMMWORD PTR [rsi+r8*4+0x50]
|         vpaddd  ymm0,ymm0,YMMWORD PTR [rdx+r8*4+0x50]
|         vmovdqu YMMWORD PTR [r9+r8*4+0x50],ymm0
|         vmovdqu ymm0,YMMWORD PTR [rsi+r8*4+0x70]
|         vpaddd  ymm0,ymm0,YMMWORD PTR [rdx+r8*4+0x70]
|         vmovdqu YMMWORD PTR [r9+r8*4+0x70],ymm0
|         add edi,0x20
|         cmp edi,r11d
+         jl L0
```

[note] 1 add - 1 array integer add

Sum(array[], const)

```
@Param({"5"})
public Integer anInt;

private int[] array;

@Setup(Level.Trial)
public void setUp() {
    this.array = new int[100_000_000];
    for (int i = 0; i < array.length; i++)
        array[i] = i + 1;
}


@Benchmark
public int [] hotMethod() {
    for (int i = 0; i < array.length; i++) {
        array[i] = array[i] + anInt.intValue();
    }
    return array;
}
```

Sum(array[], const)

JIT C2

vectorized main loop
loop unrolling
- 8x8 adds per cycle -

```
+ L0:   vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0x10]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0x10],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0x30]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0x30],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0x50]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0x50],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0x70]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0x70],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0x90]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0x90],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0xb0]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0xb0],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0xd0]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0xd0],ymm0
|       vmovdqu ymm0,YMMWORD PTR [r10+rcx*4+0xf0]
|       vpaddd  ymm0,ymm0,ymm1
|       vmovdqu YMMWORD PTR [r10+rcx*4+0xf0],ymm0
|       add ecx,0x40
|       cmp ecx,r8d
+       jl L0
```

[note] 1 add - 1 array integer add

JIT Vectorization Loop

scalar pre loop

e.g. alignment (CPU cache)

vectorized main loop

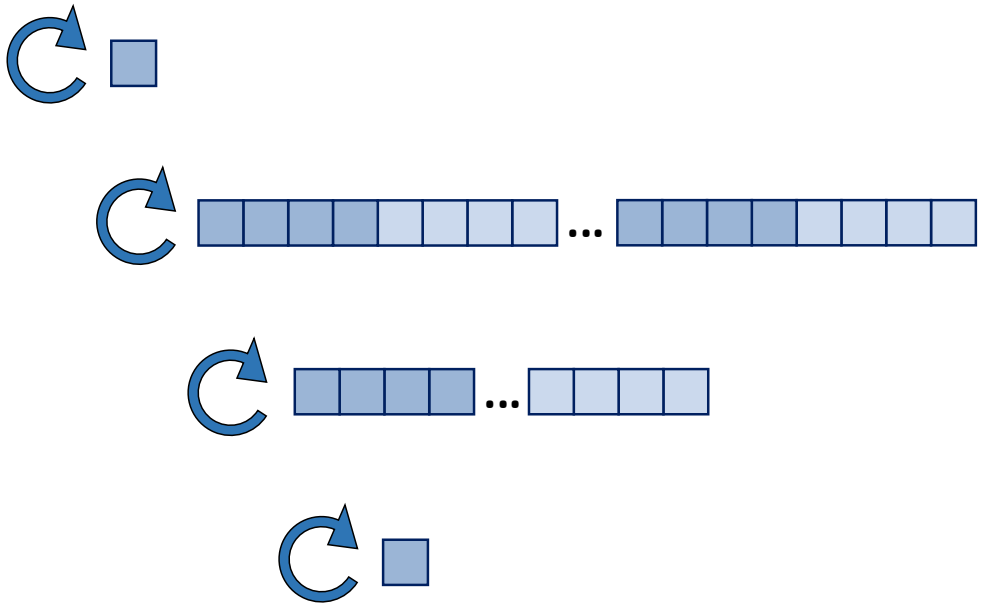
e.g. 8x8 integer adds per cycle

vectorized post loop

e.g. 8 integer adds per cycle

scalar post loop

e.g. 1 integer add per cycle



[note] pattern valid for JDK 9

Benchmark comparison

Test	Array Size	JIT C2 ms/op	LLVM Clang ms/op
Sum(array[], array[])	10^9	61.9	62.4
Sum(array[], const)	10^9	30.0	31.6

lower is better

No difference in performance at all!

Sequential sum of N integers

$$\left(\sum_{i=1}^{N} i\right)$$

Sequential sum (N integers)

Source code

```
@Benchmark
public long hotMethod() {
    long sum = 0;

    for (int i = 1; i <= N; i++) {
        sum += i;
    }

    return sum;
}
```

Sequential sum (N integers)

JIT C2

```
                mov r10d,ebx
                inc r10d

+L0:            movsxd r11,ebx
|               add rbp,r11
|               inc ebx
|               cmp ebx,r10d
+               jl L0
                jmp L2

+L1: mov        rbp,rax
|               L2: movsxd r10,ebx
|               add rbp,r10
|               ...
|               mov rax,r10
|               add rax,rbp
|               add rax,r10
|               ...
|               add rax,0x78
|               add ebx,0x10
|               cmp ebx,0x5f5e0f2
+               jl L1

+L3:            movsxd r10,ebx
|               add rax,r10
|               inc ebx
|               cmp ebx,0x5f5e101
+               jl L3
```

Sequential sum (N integers)

JIT C2

scalar pre loop
- 1 integer add per cycle -

```
                mov r10d,ebx
                inc r10d

+L0:            movsxd r11,ebx
|               add rbp,r11
|               inc ebx
|               cmp ebx,r10d
+               jl L0
                jmp L2

+L1: mov        rbp,rax
|               L2: movsxd r10,ebx
|               add rbp,r10
|               ...
|               mov rax,r10
|               add rax,rbp
|               add rax,r10
|               ...
|               add rax,0x78
|               add ebx,0x10
|               cmp ebx,0x5f5e0f2
+               jl L1

+L3:            movsxd r10,ebx
|               add rax,r10
|               inc ebx
|               cmp ebx,0x5f5e101
+               jl L3
```

Sequential sum (N integers)

JIT C2

scalar pre loop
- 1 integer add per cycle -

```
+L0:    movsxd r11,ebx
|       add rbp,r11
|       inc ebx
|       cmp ebx,r10d
+       jl L0
        jmp L2
```

scalar main loop
loop unrolling
- 16 integer adds per cycle -

```
+L1: mov rbp,rax
|    L2: movsxd r10,ebx
|    add rbp,r10
|    ...
|    mov rax,r10
|    add rax,rbp
|    add rax,r10
|    ...
|    add rax,0x78
|    add ebx,0x10
|    cmp ebx,0x5f5e0f2
+    jl L1
```

```
+L3:    movsxd r10,ebx
|       add rax,r10
|       inc ebx
|       cmp ebx,0x5f5e101
+       jl L3
```

Sequential sum (N integers)

JIT C2

scalar pre loop
- 1 integer add per cycle -

```
mov r10d,ebx
inc r10d

+L0:    movsxd r11,ebx
|       add rbp,r11
|       inc ebx
|       cmp ebx,r10d
+       jl L0
        jmp L2
```

scalar main loop
loop unrolling
- 16 integer adds per cycle -

```
+L1: mov rbp,rax
|    L2: movsxd r10,ebx
|       add rbp,r10
|       ...
|       mov rax,r10
|       add rax,rbp
|       add rax,r10
|       ...
|       add rax,0x78
|       add ebx,0x10
|       cmp ebx,0x5f5e0f2
+       jl L1
```

scalar post loop
- 1 integer add per cycle -

```
+L3:    movsxd r10,ebx
|       add rax,r10
|       inc ebx
|       cmp ebx,0x5f5e101
+       jl L3
```

Sequential sum (N integers)

LLVM clang

```
mov     eax, edi
dec     edi
imul    rdi, rax
shr     rdi
add     rdi, rax
mov     rax, rdi
```

Reduction formula: $\frac{N * (N-1)}{2} + N$

Sequential sum (N integers)

LLVM clang

no loop
- induction variable optimization -

```
mov     eax, edi
dec     edi
imul    rdi, rax
shr     rdi
add     rdi, rax
mov     rax, rdi
```

Reduction formula: $\frac{N * (N-1)}{2} + N$

Sequential sum (N integers)

Benchmark comparison

N size	JIT C2 ms/op	LLVM Clang ms/op
10^6	0.3	2×10^{-6}
10×10^6	3.1	2×10^{-6}
100×10^6	31.1	2×10^{-6}

lower is better

Sequential sum (N integers)

Conclusions

LLVM clang is able to recognize this pattern and to optimize it to a *reduction formula*, hence the throughput is almost constant

Runtime performance is not dependent on size of **N**.

Optimizations done by JIT C2 (e.g. loop peeling, loop unrolling) do not bring too much of improvements in comparison to LLVM clang.

Runtime performance is dependent on size of **N**.

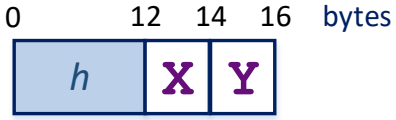
Sequential sum (N integers)

Hypothetically, **JIT** could optimize the whole loop into a reduction formula (e.g. $\frac{N*(N-1)}{2}$), but **it would need more advanced loop analysis** (e.g. something akin to LLVM's *Scalar Evolution and Loop Optimization* - to recognize how sum is related to the *loop induction variable*) **which at the moment is not happening in JIT**.

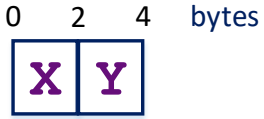
Fields Layout

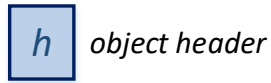
Fields Layout

```
class Vanilla {
    short x;
    short y;
}
```



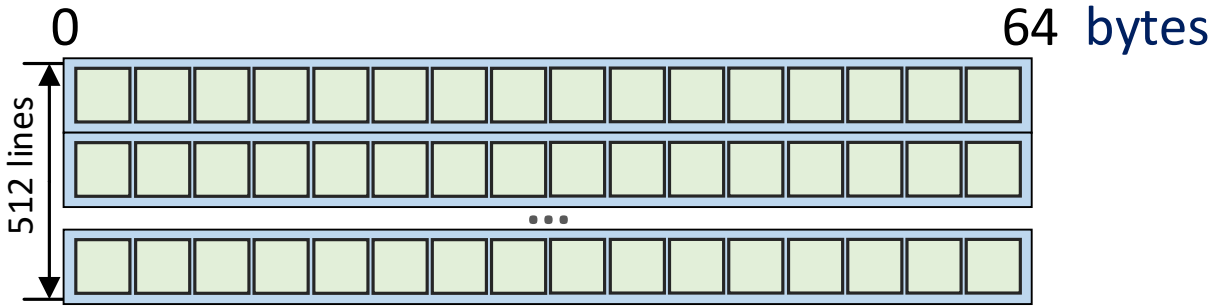

```
struct Vanilla {
    short x;
    short y;
}
```





[note] object header for 64-bit / using compressed OOPs / 8 bytes aligned

Fields Layout



L1 CPU Cache Lines

	# of objects fits Cache Line (Java class – 16B)	**# of objects fits Cache Line** (C++ struct – 4B)
Vanilla	**4**	**16**

Fields Layout

```
@Benchmark
@Group("Vanilla")
public double testVanilla(VanillaState state) {
    long sum = 0;
    for (int i = 0; i < state.array.length; i++) {
        sum += (state.array[i]).x + (state.array[i]).y; // vanilla.x + vanilla.y
    }
    return sum;
}
```

```
/** array.length = 2048 **/
```

Fields Layout

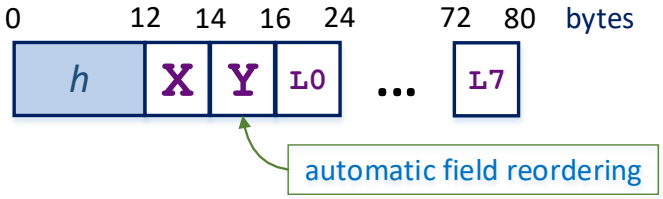
Benchmark comparison

	JIT C2 us/op	**LLVM Clang** us/op
Vanilla	1.32	1.18

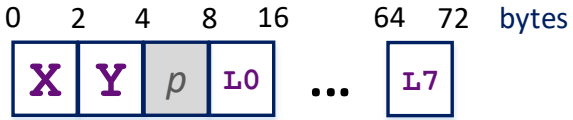
lower is better

Fields Layout

```
class Padding {
    short x;
    long l0, l1, l2, l3, l4, l5, l6, l7;
    short y;
}
```

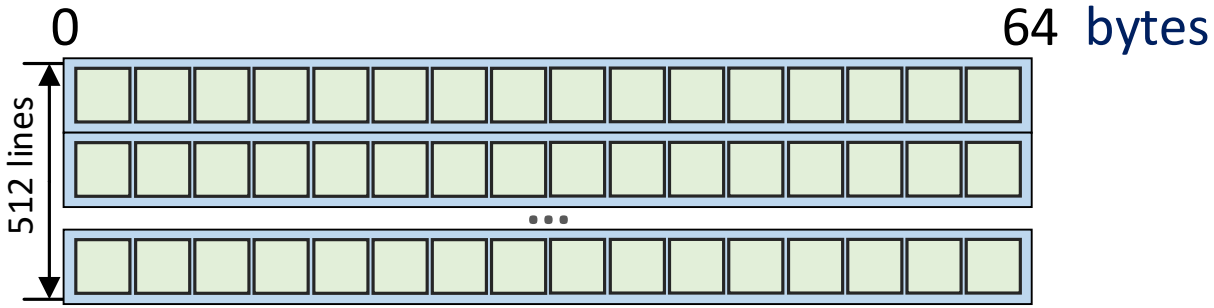



```
struct Padding {
    short x;
    short y;
    long l0, l1, l2, l3, l4, l5, l6, l7;
}
```



[note] object header for 64-bit / using compressed OOPs / 8 bytes aligned

Fields Layout



L1 CPU Cache Lines

	# of objects fits Cache Line (Java class – 80B)	**# of objects fits Cache Line** (C++ struct – 72B)
Padding	**1** *(**X** and **Y** sit on the same cache line)*	**1** *(**X** and **Y** sit on the same cache line)*

Fields Layout

```
@Benchmark
@Group("Padding")
public double testPadding(PaddingState state) {
    long sum = 0;
    for (int i = 0; i < state.array.length; i++) {
        sum += (state.array[i]).x + (state.array[i]).y; // padding.x + padding.y
    }
    return sum;
}
```

```
/** array.length = 2048 **/
```

Fields Layout

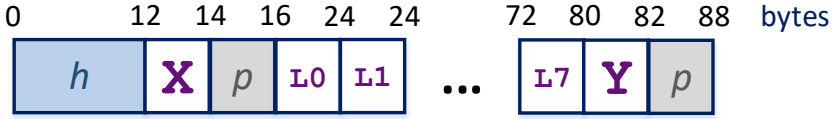
Benchmark comparison

	JIT C2 us/op	**LLVM Clang** us/op
Padding	1.74	1.73

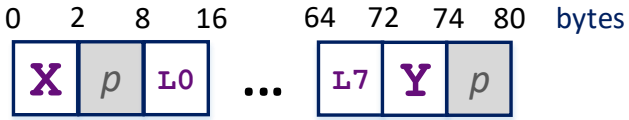
lower is better

Fields Layout

```
class Base {
    short x;
    long l0, l1, l2, l3, l4, l5, l6, l7;
}
class Hierarchy extends Base {
    short y;
}
```

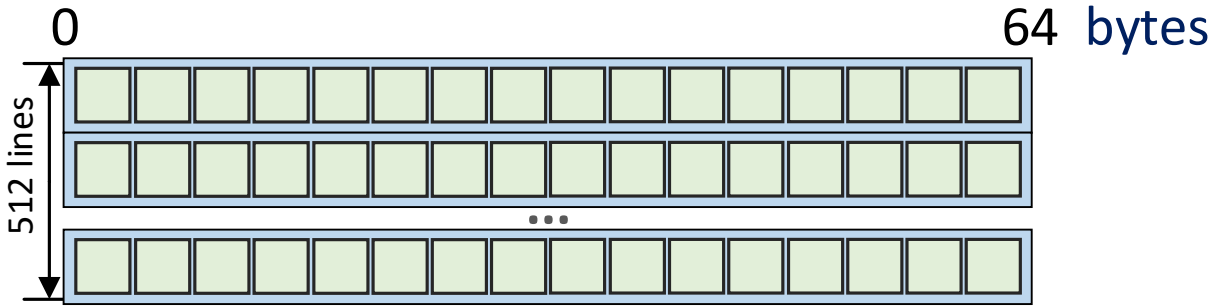



```
struct Base {
    short x;
    long l0, l1, l2, l3, l4, l5, l6, l7;
}
struct Hierarchy: Base {
    short y;
}
```



[note] object header for 64-bit / using compressed OOPs / 8 bytes aligned

Fields Layout



L1 CPU Cache Lines

	# of objects fits Cache Line (Java class – 88B)	**# of objects fits Cache Line** (C++ struct – 80B)
Hierarchy	**1** *(**X** and **Y** sit on different cache lines)*	**1** *(**X** and **Y** sit on different cache lines)*

Fields Layout

```
@Benchmark
@Group("Hierarchy")
public double testHierarchy(HierarchyState state) {
    long sum = 0;
    for (int i = 0; i < state.array.length; i++) {
        sum += (state.array[i]).x + (state.array[i]).y; // hierarchy.x + hierarchy.y
    }
    return sum;
}
```

```
/** array.length = 2048 **/
```

Fields Layout

Benchmark comparison

	JIT C2 us/op	**LLVM Clang** us/op
Hierarchy	1.96	1.93

lower is better

Fields Layout

Conclusions

Java automatically does fields reordering, nevertheless the object header impacts dCache.

LLVM clang does not automatically do fields reordering (i.e. it is at the mercy of developers), however C++ does not have the overhead of object header.

In case of C++ (e.g. LLVM clang) a bad memory alignment can hurts performance, instead Java comes with automatically fields reordering which might help.

Cases where fields reordering helps and where we should pay attention!

Fields Reordering

Useful

- **improves the overall object size** by minimizing the object layout
- **prevents unaligned** fields **access** (i.e. objects - 8 bytes alignment / fields – type aligned)
- **re-grouping fields** so they might fall on the same cache line, hence **minimizing CPU cache misses** on fields access

Fields Reordering

Useful

- **improves the overall object size** by minimizing the object layout
- **prevents unaligned** fields **access** (i.e. objects - 8 bytes alignment / fields – type aligned)
- **re-grouping fields** so they might fall on the same cache line, hence **minimizing CPU cache misses** on fields access

Pay Attention

- **false-sharing** problem
- **re-grouping fields** might also be problematic when they end up in falling on different cache lines, **maximizing CPU cache misses**

Null Checks

Null Checks

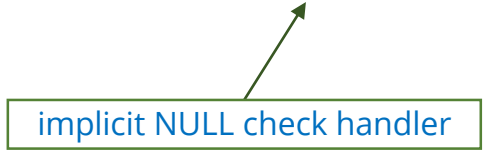
Source code

```
@Benchmark
public int hotMethod(Integer anInt) {  // anInt != NULL
    int counter = 0;

    if (null != anInt)
        counter = anInt * 42;

    return counter;
}
```

Null Checks

JIT C2

```
mov     r10d,DWORD PTR [rsi+0xc]              ; *getfield anInt
imul    eax,DWORD PTR [r12+r10*8+0xc],0x2a    ; 0x2a = 42
                                              ; implicit exception: dispatches to 0x00007f02e038c062
```



Null Checks

JIT C2

```
mov     r10d,DWORD PTR [rsi+0xc]              ; *getfield anInt
imul    eax,DWORD PTR [r12+r10*8+0xc],0x2a    ; 0x2a = 42
                                              ; implicit exception: dispatches to 0x00007f02e038c062
```

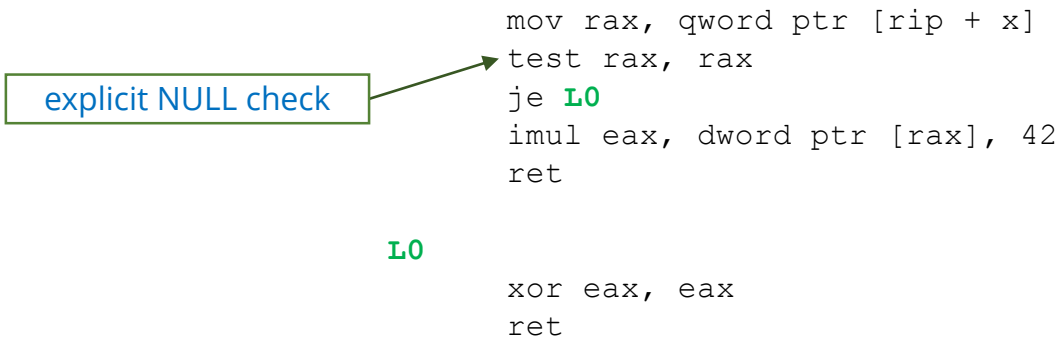
implicit NULL check handler

```
0x00007f02e038c062: mov     esi,0xffffff5d
                    mov     DWORD PTR [rsp],r10d
                    call    0x00007f02e0317c00  ; {runtime_call UncommonTrapBlob}
                    call    0x00007f02e4bae300  ; *if_acmpeq {reexecute=0 rethrow=0 return_oop=0}
                                                ; - com.jpt.NullSanityChecks::nullCheck@4
```

uncommon trap

Null Checks

LLVM clang



Null Checks

Benchmark comparison

JIT C2 ns/op	**LLVM Clang** ns/op
2.3	2

lower is better

Explicit NULL check in this particular test case does not make a big difference between these two approaches on modern hardware.

[note] could not find any way for C++ Google benchmark to have ns decimal precision

Null Checks

Conclusions

LLVM clang adds an explicit NULL check (i.e. hotMethod() has the explicit check).

JIT C2 relies on signals (e.g. SEG_FAULT) natively provided by hardware and handled by OS to manage NULLs, it optimizes without adding the NULL check .

What happens if hotMethod() is called with both NULL and !NULL ?

Null Checks

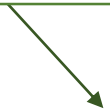
Source code

```
@Benchmark
public int hotMethod(Integer anInt) {   // anInt == NULL || anInt != NULL
    int counter = 0;

    if (null != anInt)
        counter = anInt * 42;

    return counter;
}
```

Null Checks

JIT C2

explicit NULL check



```
        mov     r11d,DWORD PTR [rsi+0xc]            ; *getfield anInt
        test    r11d,r11d
        je      L0
        imul    eax,DWORD PTR [r12+r11*8+0xc],0x2a ; 0x2a = 42
        jmp     L1

L0      xor     eax,eax
L1      add     rsp,0x10
```

Not any uncommon trap, JIT falls back to a classic way of handling NULLs (similar to LLVM clang)

Null Checks

Benchmark comparison

JIT C2 ns/op	**LLVM Clang** ns/op
2.5	2

lower is better

Still no evident difference in performance at all!

Null Checks

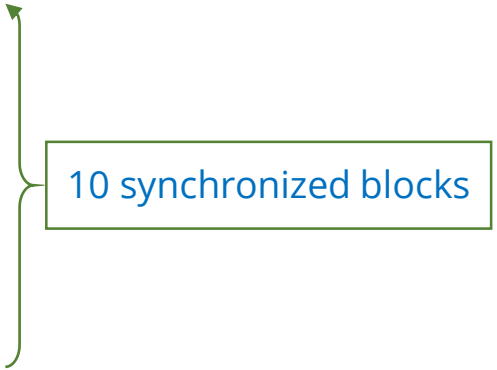
When the **number of well-predictable branches** within a code region **goes beyond the hardware limit** (i.e. total amount of branch predictor resource is finite), branch prediction starts falling off the cliff and **there might be a significant difference between these two**.

Lock Elision

Lock Elision

Source code

```
public int hotMethod() {
    int sum = 0;

    Object lock = new Object();

    synchronized (lock) {
        sum += value;
    }

   // ...

    synchronized (lock) {
        sum += value;
    }

    return sum;
}
```



Lock Elision

JIT C2

```
mov     r11d,DWORD PTR [rsi+0xc]  ;*getfield value
mov     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
```

Lock Elision

JIT C2

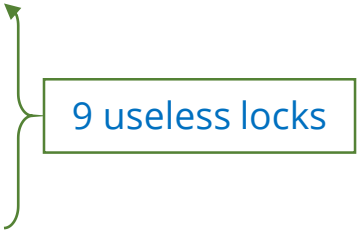
sequential sum
- no locks in between -

```
mov     r11d,DWORD PTR [rsi+0xc]  ;*getfield value
mov     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
add     eax,r11d
```

Lock Elision

LLVM clang

```
mov r14d, edi

call std::__1::mutex::lock()
mov rdi, rbx
call std::__1::mutex::unlock()

// ...

mov rdi, rbx
call std::__1::mutex::lock()
add r14d, r14d                  ; r14d <- r14d + r14d
lea ebp, [r14 + 4*r14]          ; ebp <- [r14 + 4 x r14]
mov rdi, rbx
call std::__1::mutex::unlock()
```



Lock Elision

LLVM clang

```
mov r14d, edi

call std::__1::mutex::lock()
mov rdi, rbx
call std::__1::mutex::unlock()

// ...

mov rdi, rbx
call std::__1::mutex::lock()
add r14d, r14d                  ; r14d <- r14d + r14d
lea ebp, [r14 + 4*r14]          ; ebp <- [r14 + 4 x r14]
mov rdi, rbx
call std::__1::mutex::unlock()
```

9 useless locks

compute sum inside last (10th) synchronized block

Lock Elision

Benchmark comparison

JIT C2 ns/op	**LLVM Clang** ns/op
2.8	190

lower is better

Lock Elision

Conclusions

LLVM clang cannot optimize the locks, however the additions are done inside last synchronized block.

Runtime performance is dependent on number of locks.

Java's built-in synchronized semantics are well-defined by the language hence JIT C2 takes advantage of them and remove locks which does not escape offering better performance.

Runtime performance is not dependent on number of locks.

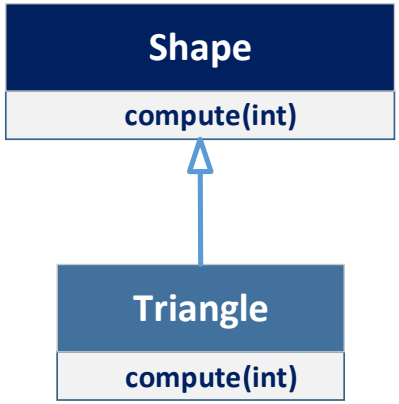
Virtual Calls

Virtual Calls

Source code

```
@State(Scope.Thread)
public static class State {
    Shape shape;

    @Setup
    public void setup() {
        shape = new Triangle();
    }
}

@Benchmark
@Group("Monomorphic")
public int testCompute(State state) {
    return compute(state.shape, param); // return (3 * 17)
}
```

```
@Param({"3"})
public int param;

public int compute(Shape shape, int i){
    return shape.compute(i);  // return (param * i)
}
```



Virtual Calls

JIT C2

```
mov     eax,ecx
shl     eax,0x4         ; eax <- eax x 16
add     eax,ecx         ; e.g. param x 17
```

Single target call (e.g. monomorphic call) is inlined without any uncommon trap.

Virtual Calls

LLVM clang

```
mov rax, qword ptr [rip + (anonymous namespace)::param]
mov eax, dword ptr [rax]
mov esi, eax
shl esi, 4          ; esi <- esi x 16
add esi, eax        ; e.g. param x 17
```

Virtual Calls

Benchmark comparison

	JIT C2 ns/op	**LLVM Clang** ns/op
Monomorphic calls	2.5	2

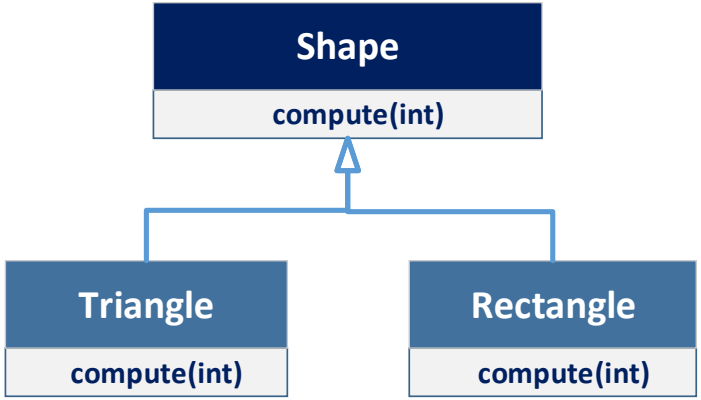
lower is better

[note] could not find any way for C++ Google benchmark to have ns decimal precision

Virtual Calls

Source code

```
@State(Scope.Thread)
public static class State {
    Shape shape1;
    Shape shape2;

    @Setup
    public void setup() {
        shape1 = new Triangle();
        shape2 = new Rectangle();
    }
}

@Benchmark
@Group("Bimorphic")
public int testCompute(State state) {
    return compute(state.shape1, param) + compute(state.shape2, param);
    // return (3 * 17) + (3 * 19)
}
```



Virtual Calls

JIT C2

```
        mov     r10d,DWORD PTR [rdx+0x8]  ; implicit exception: dispatches to 0x00007fe7cdc03991
        cmp     r10d,0xf801f06b           ; {metadata(com/jpt/Rectangle)}
        je      L0
        cmp     r10d,0xf801f02c           ; {metadata(com/jpt/Triangle)}
        jne     0x00007fe7cdc0397a        ; *invokevirtual compute {reexecute=0 rethrow=0 return_oop=0}
        mov     eax,ecx
        shl     eax,0x4
        add     eax,ecx                   ; e.g. param x 17
        jmp     L1
L0:     imul    eax,ecx,0x13              ; e.g. param x 19
L1:     add     rsp,0x20
        pop     rbp

0x00007fe7cdc0397a: call   0x00007fe7cdb8ac00 ; *invokevirtual compute {reexecute=0 rethrow=0 return_oop=0}
```

uncommon trap

Two targets call (e.g. bimorphic call) is also inlined but uncommon trap is added.

Virtual Calls

LLVM clang

```
mov rax, qword ptr [rip + (anonymous namespace)::param]
mov eax, dword ptr [rax]
shl eax, 2
lea esi, [rax + 8*rax]  ; e.g. (param x 17) + (param x 19)
```

Virtual Calls

Benchmark comparison

	JIT C2 ns/op	**LLVM Clang** ns/op
Bimorphic calls	3.4	2

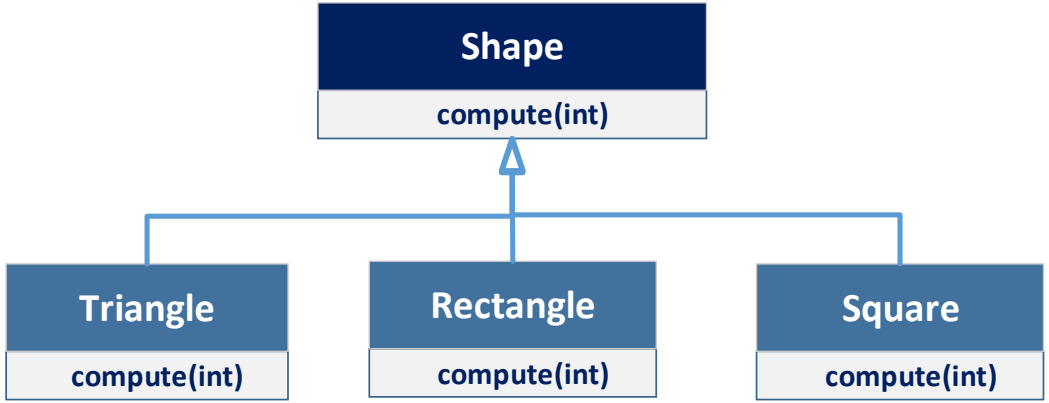
lower is better

[note] could not find any way for C++ Google benchmark to have ns decimal precision

Virtual Calls

Source code

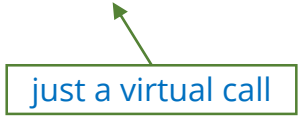
```
@State(Scope.Thread)
public static class State {
    Shape shape1;
    Shape shape2;
    Shape shape3;

    @Setup
    public void setup() {
        shape1 = new Triangle();
        shape2 = new Rectangle);
        shape3 = new Square();
    }
}

@Benchmark
@Group("Megamorphic")
public int testCompute(State state) {
    return compute(state.shape1, param) + compute(state.shape2, param)+ compute(state.shape3, param);
    // return (3 * 17) + (3 * 19) + (3 * 23)
}
```



Virtual Calls

JIT C2

```
mov     rsi,rdx
mov     edx,ecx
mov     rax,0xffffffffffffffff
call    0x00007fe469b89500  ; *invokevirtual compute {reexecute=0 rethrow=0 return_oop=0}
```



Megamorphic call is not anymore inlined, neither any uncommon trap is added.

Virtual Calls

LLVM clang

```
mov rax, qword ptr [rip + (anonymous namespace)::param]
imul esi, dword ptr [rax], 59  ; e.g. (param x 17) + (param x 19) + (param x 23)
```

Virtual Calls

Benchmark comparison

	JIT C2 ns/op	**LLVM Clang** ns/op
Megamorphic calls	8.1	2

lower is better

[note] could not find any way for C++ Google benchmark to have ns decimal precision

Virtual Calls

Conclusions

LLVM clang **does not** use vtable to resolve the virtual calls **in this case**, it does inlining and constant folding.

Runtime performance is not dependent number of target implementations **in this case**.

JIT C2 is able to inline (e.g. monomorphic and bimorphic calls) and to add uncommon traps (e.g. bimorphic calls).

Starting with 3th implementation it just performs a virtual call → due to "polluted profile" context (i.e. method is used in many different contexts with independent operand types).

Profile pollution

Profiles (especially type profiles) are subject to pollution if the profiled code is heavily reused in ways the diverge from each other.

As a simple example, if `ArrayList.contains` is used with lists that never contain nulls, some null checks will never be taken, and the profile can reflect this. But if this routine is also used with lists that occasionally contain nulls, then the "taken" count of the null check instruction may become non-zero. This in turn may influence the compiler to check operands more cautiously, with a loss of performance for *all* uses of the method.

Polluted profiles can be mitigated by a number of means, including:

- inlining with type or value information flowing from caller context
- inlining more than one type case (UseBimorphicInlining)
- context-dependent split profiles (bug 8015416)
- hand inlining by the Java programmer (which is discouraged)
- generic type reification (when pollution comes from a type parameter; not implemented)
- other forms of online code splitting
- adding type-check bytecodes for profiled references *before* the call to the shared routine

Source [https://wiki.openjdk.java.net/display/HotSpot/MethodData#MethodData-Poll]

JDK / JDK-8015416

tier one should collect context-dependent split profiles

Details

Type:	Enhancement	Status:	OPEN
Priority:	P3	Resolution:	Unresolved
Affects Version/s:	9, 10	Fix Version/s:	13
Component/s:	hotspot		
Labels:	c1 comp-backlog performance tiered-compilation		
Subcomponent:	compiler		

Description

To avoid polluted profiles, tier one should create disjoint "split" profiles for methods that it inlines. If a method is inlined three times, it should get three fresh copies of its global profile, initialized to no types or counts. When a later tier re-optimizes the code and inlines the same method, the compiler should use the context-dependent profile in preference to the global profile, if one is available.

Scalar Replacement

Scalar Replacement

Source code

```
@Param({"3"})
private int param1;

@Param({"5"})
private int param2;


@Benchmark
public int hotMethod() {
    Wrapper w = new Wrapper(param1, param2);
    return w.x + w.y;
}

public static class Wrapper {
    public int x;
    public int y;

    public Wrapper(int value1, int value2) {
        this.x = value1;
        this.y = value2;
    }
}
```

Scalar Replacement

JIT C2

```
mov     eax,DWORD PTR [rsi+0x10]
add     eax,DWORD PTR [rsi+0xc]  ; e.g. x + y
```

Scalar Replacement

LLVM clang

```
mov rax, qword ptr [rip + (anonymous namespace)::x]
mov rcx, qword ptr [rip + (anonymous namespace)::y]
mov esi, dword ptr [rcx]
add esi, dword ptr [rax]    ; e.g. x + y
```

Scalar Replacement

Benchmark comparison

JIT C2 ns/op	**LLVM Clang** ns/op
2.2	2

lower is better

No difference in performance at all!

Scalar Replacement

Conclusions

LLVM clang does constant folding at compile time, no allocation.

JIT C2 prevents the heap memory allocations, as well.

Conclusions

LLVM clang can trigger - even ahead of time - a lot of powerful optimizations (*e.g. sum 1 to N integers, virtual calls, scalar replacement*).

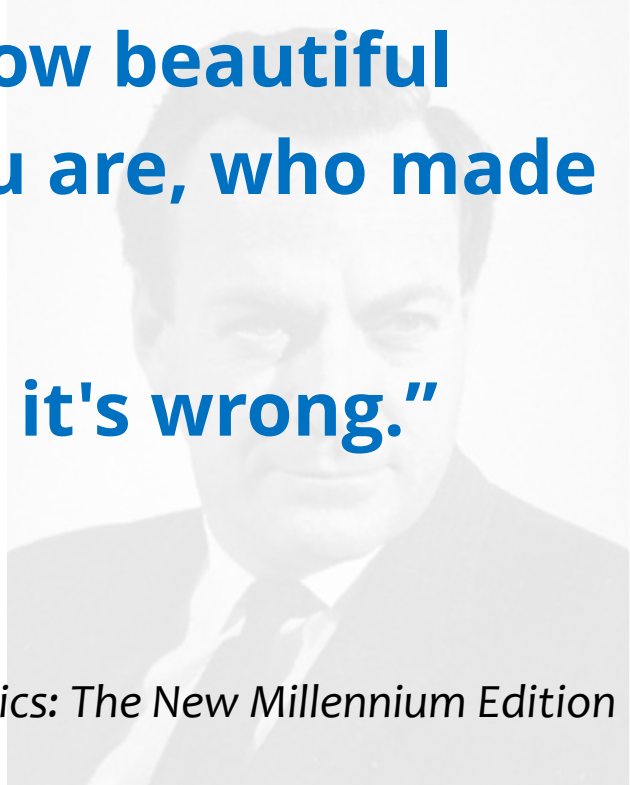
JIT C2 can do better optimizations than LLVM clang when runtime profiling information matters or built-in synchronized semantics are well-defined at language level (*e.g. lock elision*).

When both compilers optimize the code based on similar approaches (*e.g. loop unrolling/vectorization, null checks, sum(array[], array[]), sum(array[], const)*) there is almost identical performance.

In Closing …

“It doesn't make a difference how beautiful your guess is […] how smart you are, who made the guess, or what his name is.

If it disagrees with experiment, it's wrong.”



The Richard Feynman Lectures on Physics: The New Millennium Edition

Special thanks to

George Frãsineanu

C++ Senior Developer
george.frasineanu@gmail.com

THANK YOU

Ionuţ Baloşin

Java Software Architect

@ionutbalosin

LUXOFT

LXFT LISTED NYSE