A race of two compilers: Graal JIT vs. C2 JIT. Which one offers better runtime performance?

by Ionuţ Baloşin

IonutBalosin.com

@IonutBalosin

About Me

Software Architect
Technical Trainer

Raiffeisen Bank International

Java Performance Tuning

Software Architecture Essentials

Designing High-Performance Applications

< for more details please visit: IonutBalosin.com/training >

Writer
Speaker

IonutBalosin.com

InfoQ

DEVOXX VOXXED SHARE THE KNOWLEDGE Joker<?> Dev<Talks/> DevDays JBCN Conf BARCELONA JAVA USERS GROUP

ORACLE CODE geecon Riga DevDays JPRIME I.T.A.K.E. Unconference DevFest Vienna XPDAYS BENELUX geekOUT {"ON":"THE BEACH"}

Today’s Content

01 Intro

02 Scalar Replacement

03 Virtual Calls

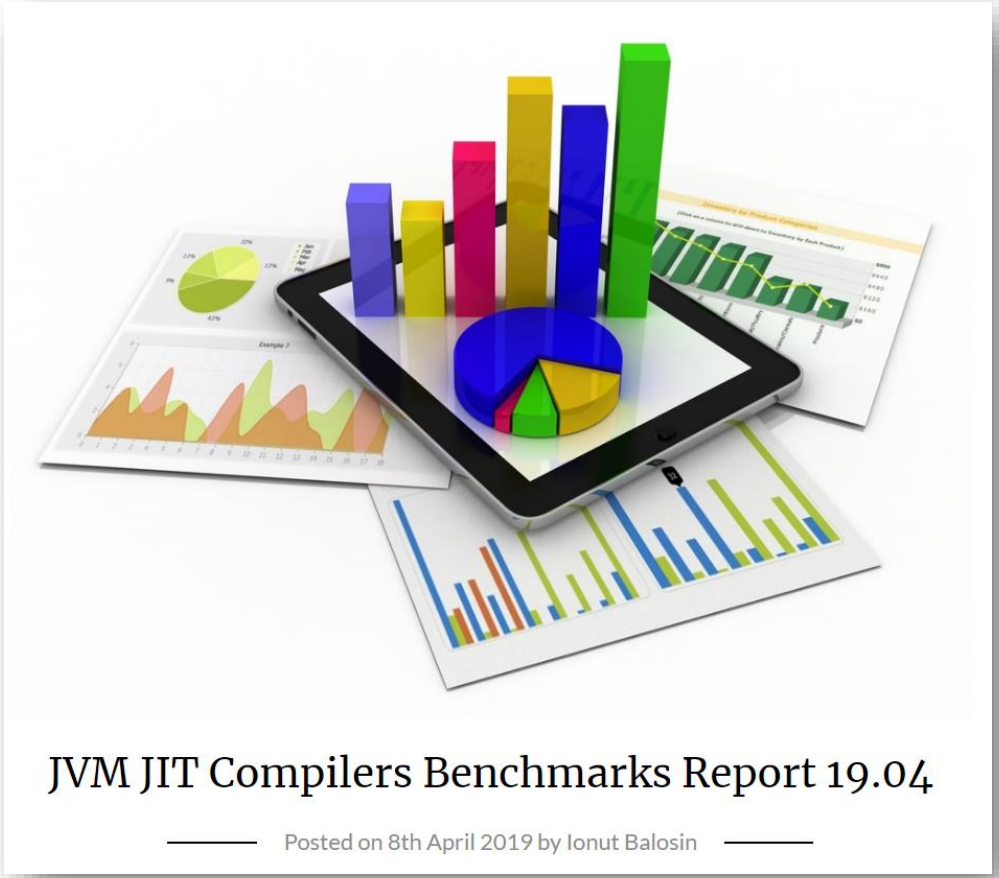
04 Vectorization & Loop Optimizations

05 Summary & Takeaways

Intro

01

Source: [IonutBalosin.com/2019/04/jvm-jit-compilers-benchmarks-report-19-04]



Configuration

CPU	Intel i7-8550U Kaby Lake R
MEMORY	16GB DDR4 2400 MHz
OS / Kernel	Ubuntu 18.10 / 4.18.0-25-generic
Java Version	13 Linux-x64
JVM	OpenJDK
JMH v1.21	5x10s warm-up iterations, 5x10s measurement iterations, 3 JVM forks, single threaded

Note: C2 JIT usually needs less warm-up iterations, being a native Compiler.

This talk is NOT about GraalVM EE !

Current optimizations might differ for other JVMs!

Fair comparison?

to

Well … it depends from the perspective:

Application/end user: **performance** and (**licensing**) **costs** are the leading factors

Compiler/internals (e.g. implementation, maturity): very different

Graal JIT

- ✓ Java implementation
- ✓ still early stage (recently become production ready; e.g. v19.2.1)
- ✓ targets JVM-based languages (e.g. Java, Scala, Kotlin), dynamic languages (e.g. JavaScript, Ruby, R), native languages (e.g. C/C++, Rust)
- ✓ modular design (JVMCI plug-in arch)
- ✓ Ahead Of Time flavor (our of scope for today's presentation)

Note: non of these compilers are in maintenance.

Graal JIT

- ✓ Java implementation
- ✓ still early stage (recently become production ready; e.g. v19.2.0.1)
- ✓ targets JVM-based languages (e.g. Java, Scala, Kotlin), dynamic languages (e.g. JavaScript, Ruby, R), native languages (e.g. C/C++, Rust)
- ✓ modular design (JVMCI plug-in arch)
- ✓ Ahead Of Time flavor (our of scope for today's presentation)

C2 JIT

- ✓ C++ implementation
- ✓ mature / production grade runtime
- ✓ written mainly to optimize Java source code
- ✓ is essentially a local maximum of performance and is reaching the end of its design lifetime
- ✓ latest improvements were more focused on intrinsics and vectorization, rather than adding new optimization heuristics

---

Note: non of these compilers are in maintenance.

Nevertheless, nowadays the baseline for Graal JIT still remains C2 JIT!

The target is to leverage its performance to be at least on par with C2 JIT especially on real world applications / production code.

Few main distinguishing factors for Graal JIT

1. Partial Escape Analysis
2. Improved inlining
3. Guard optimizations for efficient handling of speculative code

Main Graal Publications

1

Partial Escape Analysis and Scalar Replacement for Java

Lukas Stadler
Johannes Kepler University
Linz, Austria
lukas.stadler@jku.at

Thomas Würthinger
Oracle Labs
thomas.wuerthinger
@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
moessenboeck@ssw.jku.at

ABSTRACT

Escape Analysis allows a compiler to determine whether an object is accessible outside the allocating method or thread. This information is used to perform optimizations such as Scalar Replacement, Stack Allocation and Lock Elision, allowing modern dynamic compilers to remove some of the abstractions introduced by advanced programming models.

The all-or-nothing approach taken by most Escape Analysis algorithms prevents all these optimizations as soon as there is one branch where the object escapes, no matter how unlikely this branch is at runtime.

This paper presents a new, practical algorithm that performs control flow sensitive Partial Escape Analysis in a dynamic Java compiler. It allows Escape Analysis, Scalar Replacement and Lock Elision to be performed on individual

1. INTRODUCTION

State-of-the-art Virtual Machines employ techniques such as advanced garbage collection, alias analysis and biased locking to make working with dynamically allocated objects as efficient as possible. But even if allocation is cheap, it still incurs some overhead. Even if alias analysis can remove most object accesses, some of them cannot be removed. And although acquiring a biased lock is simple, it is still more complex than not acquiring a lock at all.

Escape Analysis can be used to determine whether an object needs to be allocated at all, and whether its lock can ever be contended. This can help the compiler to get rid of the object's allocation, using *Scalar Replacement* to replace the object's fields with local variables.

Escape Analysis checks whether an object escapes its al-

In a nutshell

Partial Escape Analysis (PEA) determines the escapability of objects on individual branches and reduces object allocations even if the objects escapes on some paths.

PEA works on compiler's intermediate representation not on bytecode. It is effective in conjunction with other parts of the compiler, such as *inlining*, *global value numbering*, and *constant folding*.

Example

```
class Key {
    int idx;
    Object ref;

    Key(int idx, Object ref) {
        this.idx = idx ;
        this.ref = ref ;
    }

    synchronized boolean equals(Key other) {
        return idx == other.idx && ref == other.ref ;
    }
}
```

Example from paper "*Partial Escape Analysis and Scalar Replacement for Java*".

```
static Key cacheKey;
static Object cacheValue;

// Initial code
Object getValue(int idx, Object ref) {
    Key key = new Key(idx, ref);
    if (key.equals(cacheKey)) {
        return cacheValue;
    } else {
        cacheKey = key;
        cacheValue = createValue(...);
        return cacheValue;
    }
}
```

Object is allocated

No allocation needed on this branch

```
static Key cacheKey;
static Object cacheValue;

// Code after inlining and Partial Escape Analysis
Object getValue(int idx, Object ref){
    Key tmp = cacheKey;
    if (idx == tmp.idx && ref == tmp.ref) { // equals() method was inlined
        return cacheValue;
    } else {
        Key key = alloc Key;
        key.idx = idx;
        key.ref = ref;
        cacheKey = key;
        cacheValue = createValue(...);
        return cacheValue;
    }
}
```

Object allocation is moved into the branch it escapes

2

An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers

Aleksandar Prokopec
Oracle Labs, Switzerland
aleksandar.prokopec@gmail.com

Gilles Duboscq
Oracle Labs, Switzerland
gilles.m.duboscq@oracle.com

David Leopoldseder
JKU Linz, Austria
david.leopoldseder@jku.at

Thomas Würthinger
Oracle Labs, Switzerland
thomas.wuerthinger@oracle.com

***Abstract*—Inlining is one of the most important compiler optimizations. It reduces call overheads and widens the scope of other optimizations. But, inlining is somewhat of a black art of an optimizing compiler, and was characterized as a computationally intractable problem. Intricate heuristics, tuned during countless hours of compiler engineering, are often at the core of an inliner implementation. And despite decades of research, well-established inlining heuristics are still missing.**

In this paper, we describe a novel inlining algorithm for JIT compilers that incrementally explores a program's call graph, and alternates between inlining and optimizations. We devise three novel heuristics that guide our inliner: adaptive decision thresholds, callsite clustering, and deep inlining trials. We implement the algorithm inside Graal, a dynamic JIT compiler for the HotSpot JVM. We evaluate our algorithm on a set of industry-standard benchmarks, including Java DaCapo, Scalabench, Spark-Perf, STMBench7 and other benchmarks,

mization prediction by incrementally exploring and specializing the program's call tree, and clusters the callsites before inlining. We report the following new observations:

- In many programs, there is an impedance between the logical units of code (i.e. subroutines) and the optimizable units of code (groups of subroutines). It is therefore beneficial to inline *specific clusters of callsites*, instead of a single callsite at a time. We describe how to identify such callsite clusters.
- When deciding when to stop inlining, using an *adaptive threshold function* (rather than a fixed threshold value) produces more efficient programs.
- By propagating a callsite's argument types throughout the call tree, and then performing optimizations in the entire

In a nutshell

Improved inlining strategy explores the call tree and performs **late inlining** as default instead of inlining during bytecode parsing.

Graal JIT devise three novel heuristics: *callsite clustering*, *adaptive decision thresholds*, and *deep inlining trials*.

Example (Scala code)

```
def main(args: Array[String]) {
    async { log(getStackTrace) }
    log(args)
}

def log[T](xs: Array[T]) {
    xs.foreach(println)
}
```

Example from paper "*An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers*".

Example (Scala code)

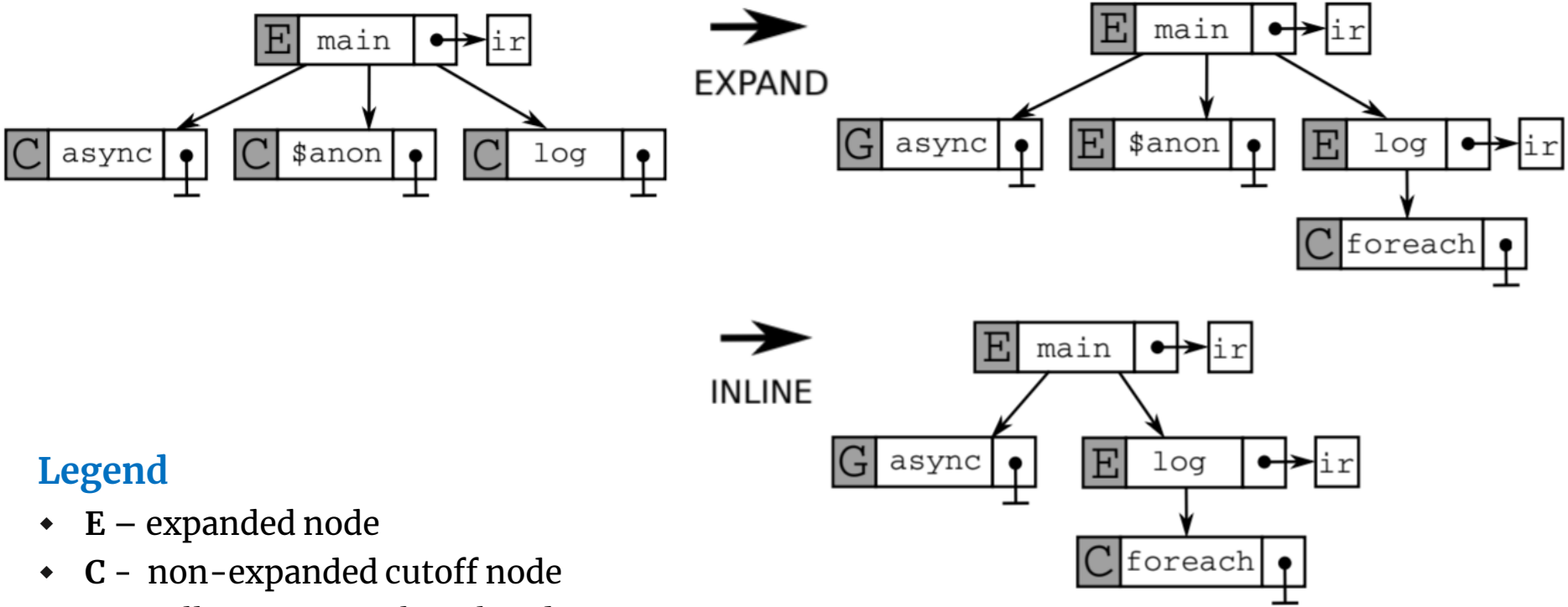
```
def main(args: Array[String]) {
    async { log(getStackTrace) }
    log(args)
}

def log[T](xs: Array[T]) {
    xs.foreach(println)
}
```

definition

```
// Scala standard library
trait IndexedSeqOptimized[T] {
    def get(i: Int): T
    def length: Int
    def foreach(f: T => Unit) {
        var i = 0
        while (i < this.length)
        { f.apply(this.get(i)); i += 1 }
    }
}
```

Note: Inlining foreach() call into log() method is only beneficial if the get() and length() calls inside the corresponding loop are also inlined.

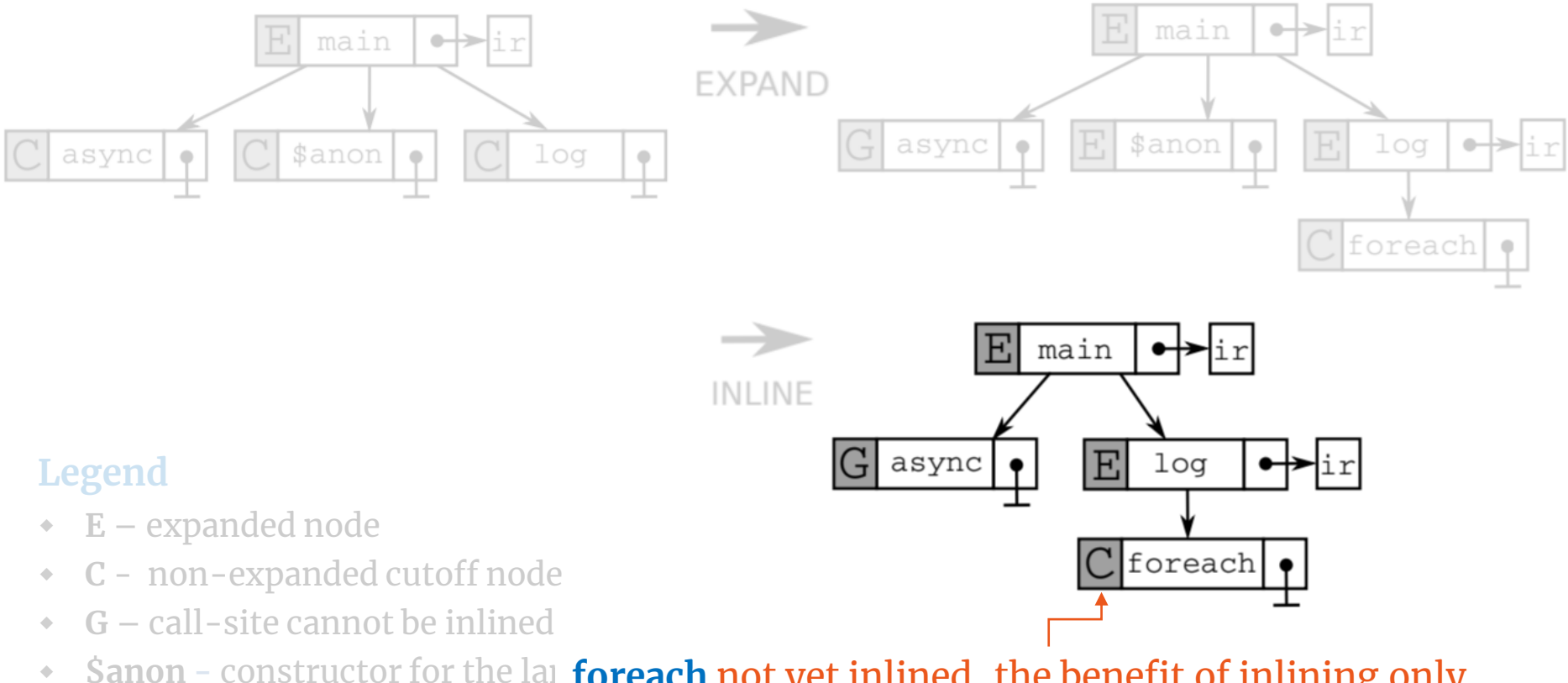
Call Tree During Inlining



Legend

- **E** – expanded node
- **C** - non-expanded cutoff node
- **G** – call-site cannot be inlined
- **$anon** - constructor for the lambda object passed to **async**

Call Tree During Inlining



Legend

- **E** – expanded node
- **C** - non-expanded cutoff node
- **G** – call-site cannot be inlined
- **$anon** - constructor for the la

foreach not yet inlined, the benefit of inlining only becomes apparent once the call tree is explored more

3

An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler

Gilles Duboscq* Thomas Würthinger† Lukas Stadler* Christian Wimmer† Doug Simon†
Hanspeter Mössenböck*

*Institute for System Software, Johannes Kepler University Linz, Austria †Oracle Labs
{duboscq, stadler, moessenboeck}@ssw.jku.at
{thomas.wuerthinger, christian.wimmer, doug.simon}@oracle.com

Abstract

We present a compiler intermediate representation (IR) that allows dynamic speculative optimizations for high-level languages. The IR is graph-based and contains nodes fixed to control flow as well as floating nodes. Side-effecting nodes include a framestate that maps values back to the original program. Guard nodes dynamically check assumptions and, on failure, deoptimize to the interpreter that continues execution. Guards implicitly use the framestate and program position of the last side-effecting node. Therefore, they can

tional optimization opportunities, because the cold branch and its influence on program state need not be taken into account when compiling a method. In high level languages such as Java, a single operation can include an implicit control-flow split. For example, a field access includes a null check on the receiver that can throw an exception. This control-flow path is not visible in the original source program, but the compiler still has to handle it. In this context, the speculative reduction of those control-flow paths is important. Dynamic languages can profit even more from this

Scalar Replacement

02

Escape Analysis (EA) analyses the scope of a new object and decides whether it might be allocated or not on the heap.

EA is not an optimization but rather an **analysis phase for the optimizer**. The result of this analysis allows compiler to perform optimizations like *object allocations*, *synchronization primitives* and *field accesses*.

Object Scopes

- ✓ **NoEscape** (local to current method/thread, not visible outside)
- ✓ **ArgEscape** (passed as an argument but not visible by other threads)
- ✓ **GlobalEscape** (visible outside of current method/thread)

For **NoEscape** objects compiler can remap the accesses to the object fields to accesses to synthetic local operands: **Scalar Replacement** optimization.

Experiment

Case I	no_escape
Case II	no_escape_object_array
Case III	branch_escape_object
Case IV	arg_escape_object

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

@Benchmark
public int no_escape() {
    LightWrapper w = new LightWrapper(param);
    return w.f1 + w.f2;
}
public static class LightWrapper {
    public int f1;   // param
    public int f2;   // param << 1
    //...
}
```

```
@Benchmark
public int no_escape() {
    LightWrapper w = new LightWrapper(param);
    return w.f1 + w.f2;
}
```

equivalent to

```
return f1 + f2;
```

```
@Param({"3"})   private int param;
@Param({"128"}) private int size;

@Benchmark
public int no_escape_object_array() {
    HeavyWrapper w = new HeavyWrapper(param, size);
    return w.f1 + w.f2 + w.z.length;
}
public static class HeavyWrapper {
    public int f1;   // param
    public int f2;   // param << 1
    public byte z[]; // new byte[size]
    // ...
}
```

```
@Benchmark
public int no_escape_object_array() {
    HeavyWrapper w = new HeavyWrapper(param, size);
    return w.f1 + w.f2 + w.z.length;
}
```

equivalent to

```
return f1 + f2 + length;
```

CASE III

```
@Param(value = {"false"}) private boolean objectEscapes;

@Benchmark
public void branch_escape_object(Blackhole bh) {
    HeavyWrapper wrapper = new HeavyWrapper(param, size);
    HeavyWrapper result;

    if (objectEscapes) // false
        result = wrapper;
    else
        result = globalCachedWrapper;

    bh.consume(result);
}
```

CASE III

```
@Param(value = {"false"}) private boolean objectEscapes;

@Benchmark
public void branch_escape_object(Blackhole bh) {
    HeavyWrapper wrapper = new HeavyWrapper(param, size);
    HeavyWrapper result;

    if (objectEscapes) // false
        result = wrapper;
    else
        result = globalCachedWrapper;

    bh.consume(result);
}
```

branch is never taken,
wrapper never escapes,
its scope is NoEscape

CASE III

```
@Param(value = {"false"}) private boolean objectEscapes;

@Benchmark
public void branch_escape_object(Blackhole bh) {
    HeavyWrapper wrapper = new HeavyWrapper(param, size);
    HeavyWrapper result;

    if (objectEscapes) // false
        result = wrapper;
    else
        result = globalCachedWrapper;

    bh.consume(result);
}
```

branch is never taken, wrapper never escapes, its scope is NoEscape

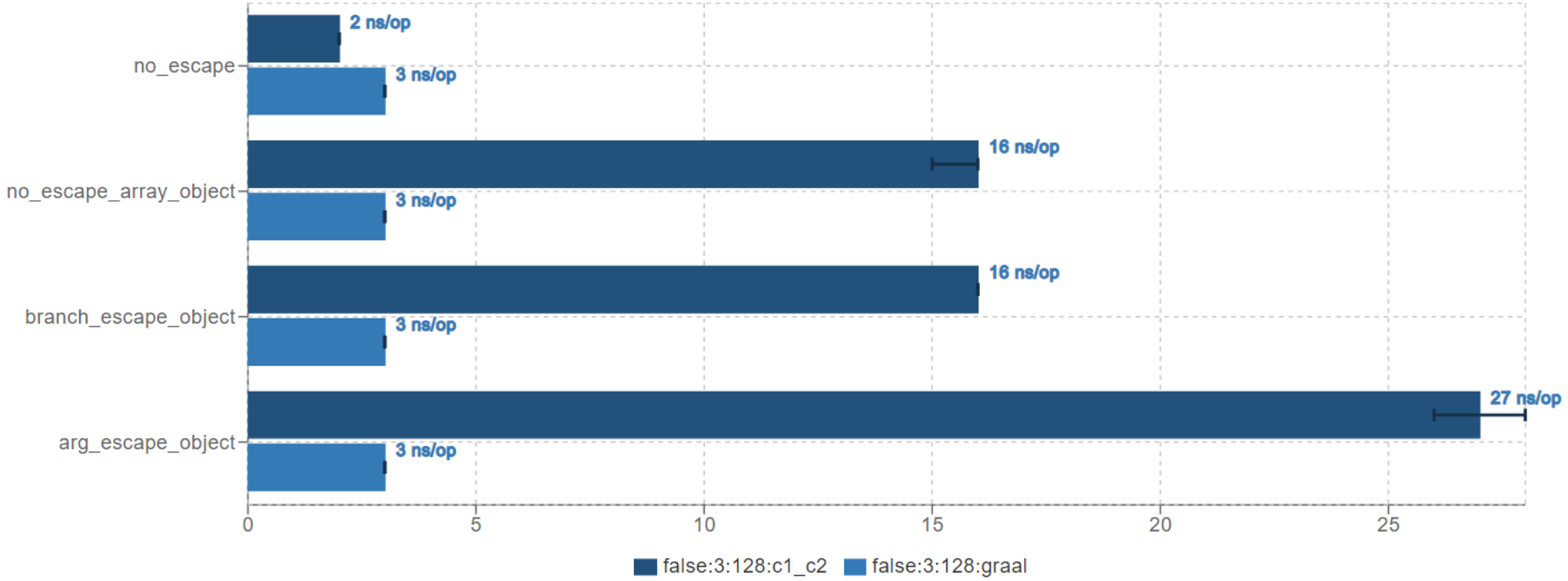
Based on Partial Escape Analysis, the wrapper object allocation is eliminated.

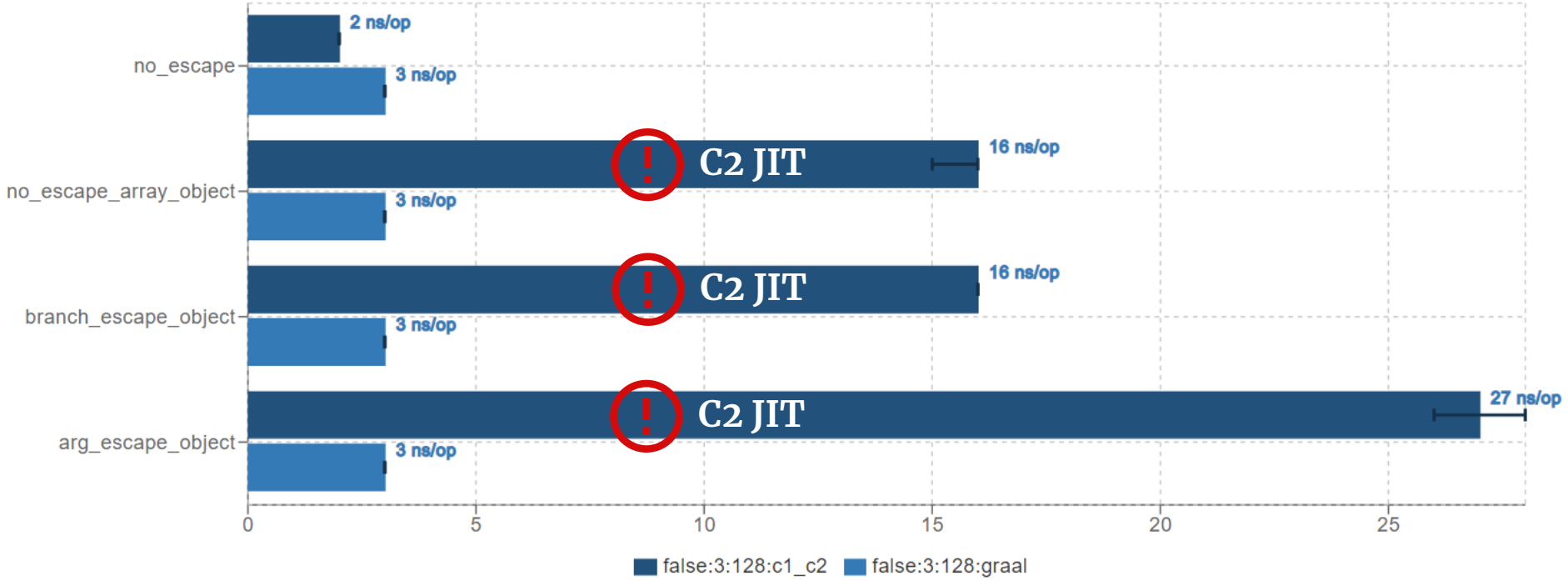
```
@Benchmark
public boolean arg_escape_object() {
    HeavyWrapper wrapper1 = new HeavyWrapper(param, size);
    HeavyWrapper wrapper2 = new HeavyWrapper(param, size);
    boolean match = false;

    if (wrapper1.equals(wrapper2))
        match = true;

    return match;
}
```

```
@Benchmark
public boolean arg_escape_object() {
    HeavyWrapper wrapper1 = new HeavyWrapper(param, size);
    HeavyWrapper wrapper2 = new HeavyWrapper(param, size);
    boolean match = false;

    if (wrapper1.equals(wrapper2))
        match = true;

    return match;
}
```

wrapper1 scope is NoEscape

wrapper2 scope is:
- NoEscape, if inlining of equals() succeeds
- ArgEscape, if inlining fails or disabled


2 ns/op
no_escape
3 ns/op
16 ns/op
no_escape_array_object
3 ns/op
16 ns/op
branch_escape_object
3 ns/op
27 ns/op
arg_escape_object
3 ns/op
0
5
10
15
20
25
false:3:128:c1_c2
false:3:128:graal


2 ns/op
no_escape
3 ns/op
C2 JIT
16 ns/op
no_escape_array_object
3 ns/op
C2 JIT
16 ns/op
branch_escape_object
3 ns/op
C2 JIT
27 ns/op
arg_escape_object
3 ns/op
0
5
10
15
20
25
false:3:128:c1_c2
false:3:128:graal

Conclusions

Graal JIT was able to get rid of heap allocations offering a constant response time, around 3 ns/op

Conclusions

Graal JIT was able to get rid of heap allocations offering a constant response time, around 3 ns/op

C2 JIT struggles to get rid of the heap allocations in case:

- there is a condition which does not make obvious at compile time if the object escapes
- or if the object scope becomes local (i.e. NoEscape) after inlining

Conclusions

Graal JIT was able to get rid of heap allocations offering a constant response time, around 3 ns/op

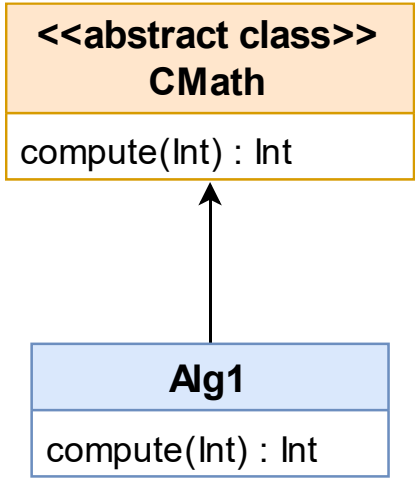
C2 JIT struggles to get rid of the heap allocations in case:
- there is a condition which does not make obvious at compile time if the object escapes
- or if the object scope becomes local (i.e. NoEscape) after inlining

Note: C2 JIT by default does not consider escaping arrays if their size (i.e. the number of elements) is greater than 64

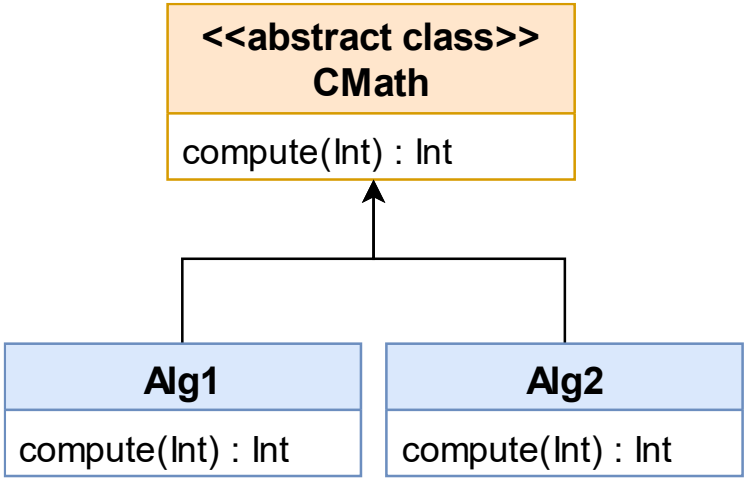
-XX:EliminateAllocationArraySizeLimit=<arraySize>

Virtual Calls

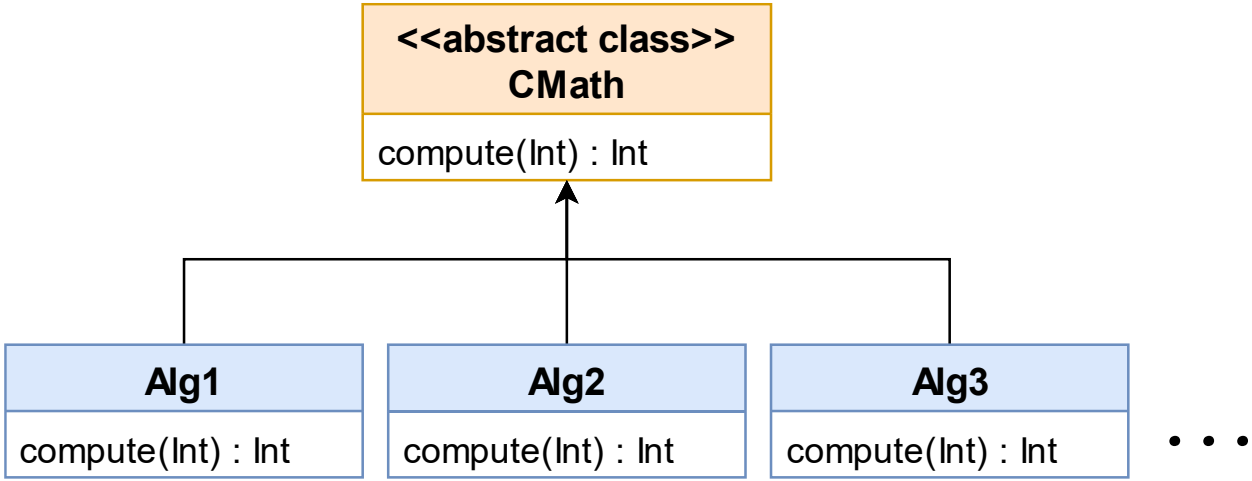
03



Monomorphic call site – optimistically points to **only one concrete method** that has ever been used at that particular call site.



Bimorphic call site – optimistically points **to only two concrete methods** which can be invoked at a particular call site.



Megamorphic call site – optimistically points to **three or possibly more methods** which can be invoked at a particular call site.

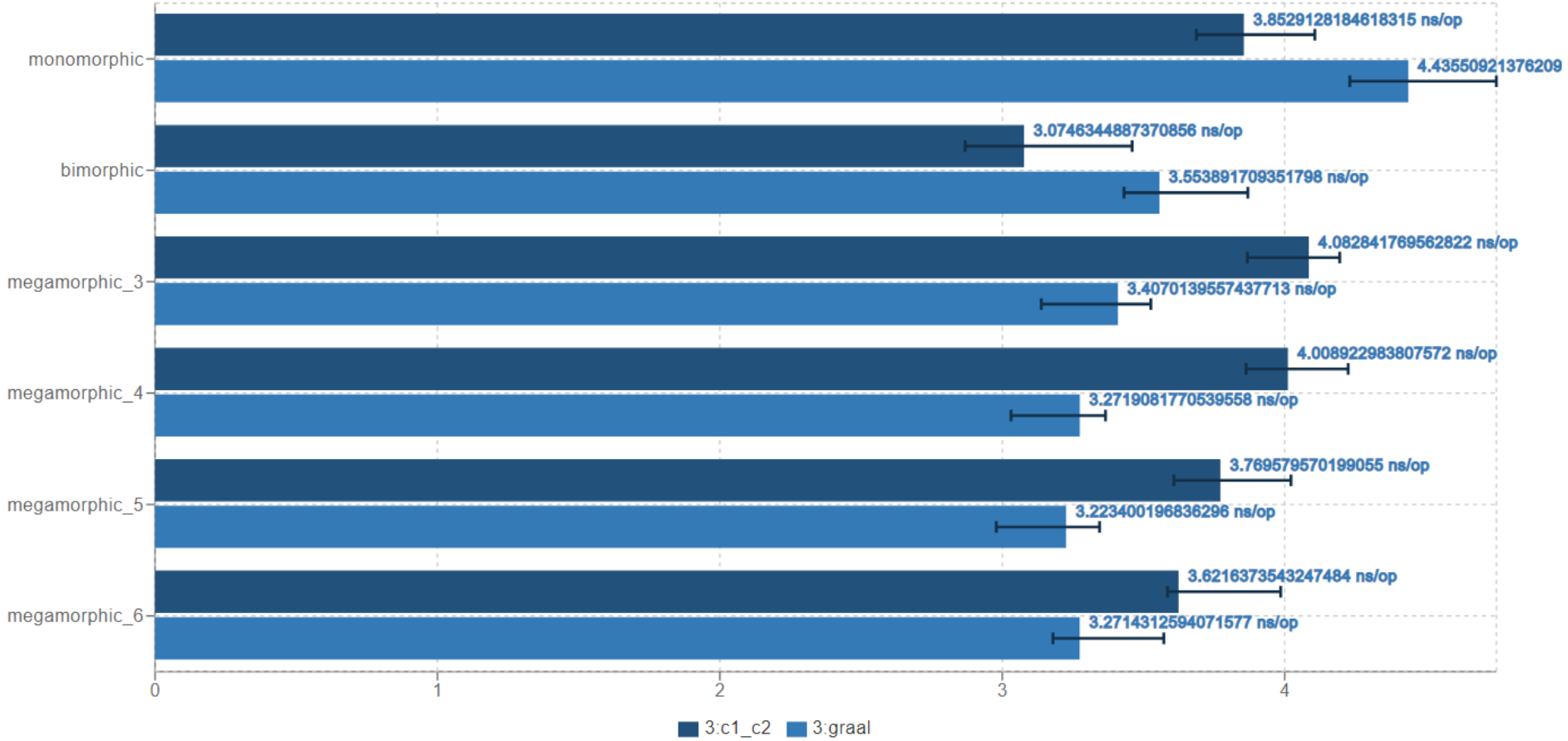
```
// CMath = {Alg1, Alg2, Alg3, Alg4, Alg5, Alg6}
@CompilerControl(CompilerControl.Mode.DONT_INLINE)
public static int execute(CMath cmath, int i) {
    return cmath.compute(i); // param * constant
}
```

megamorphic call site

Note: check the annex for another approach of testing the megamorphic call site

```
@Benchmark
@OperationsPerInvocation(1)
public int monomorphic() {
    return execute(alg1, param)    // param * 17
}

@Benchmark
@OperationsPerInvocation(2)
public int bimorphic() {
    return execute(alg1, param) + // param * 17
           execute(alg2, param)    // param * 19
}
```

```
@Benchmark
@OperationsPerInvocation(3)
public int megamorphic_3() {
    return execute(alg1, param) + // param * 17
           execute(alg2, param) + // param * 19
           execute(alg3, param)   // param * 23
}

/*
  Similar test cases for:
    megamorphic_4() {...}
    megamorphic_5() {...}
    megamorphic_6() {...}
*/
```


monomorphic
bimorphic
megamorphic_3
megamorphic_4
megamorphic_5
megamorphic_6
3.8529128184618315 ns/op
4.43550921376209
3.0746344887370856 ns/op
3.553891709351798 ns/op
4.082841769562822 ns/op
3.4070139557437713 ns/op
4.008922983807572 ns/op
3.2719081770539558 ns/op
3.769579570199055 ns/op
3.223400196836296 ns/op
3.6216373543247484 ns/op
3.2714312594071577 ns/op
0
1
2
3
4
3:c1_c2
3:graal

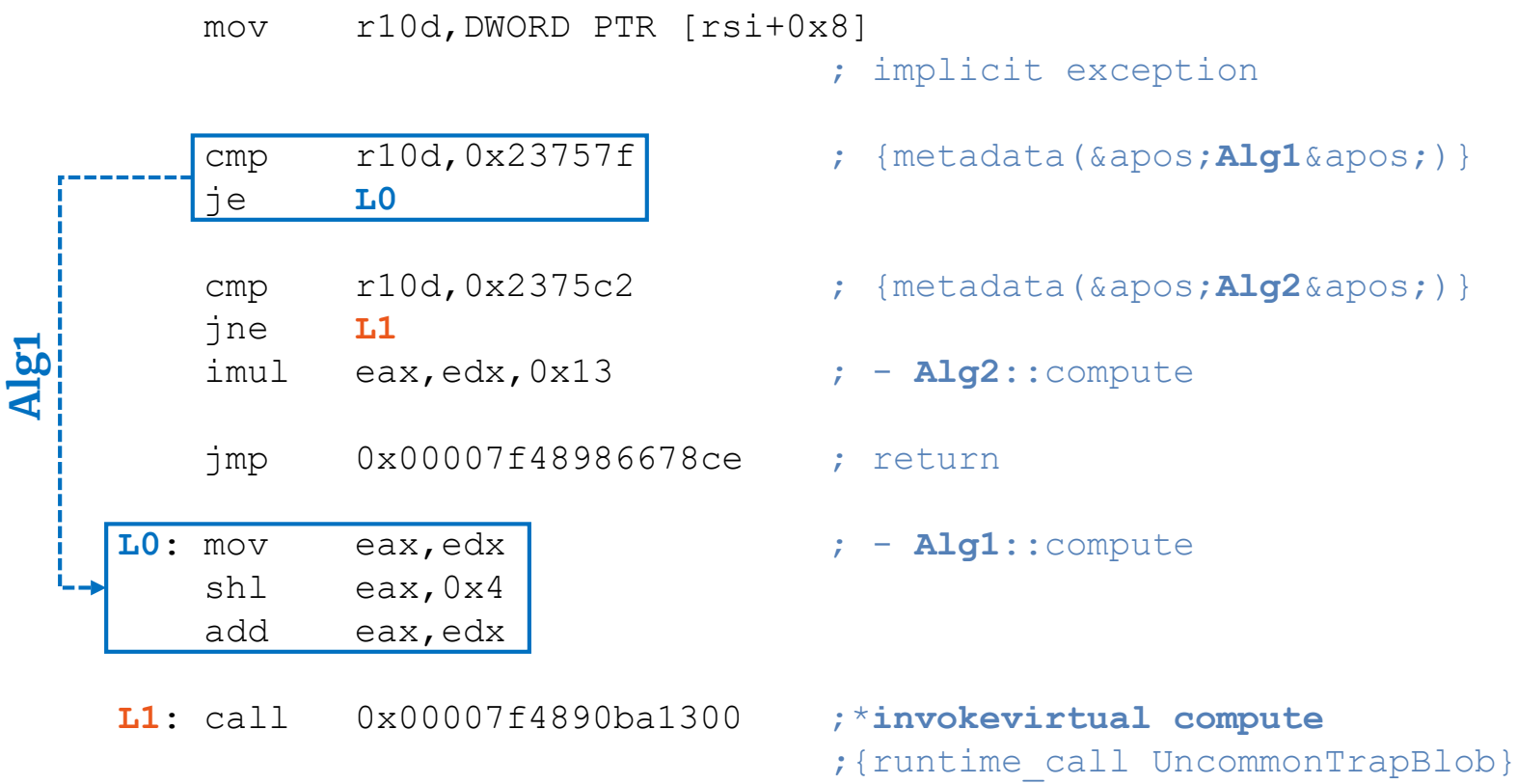

monomorphic
bimorphic
megamorphic_3
megamorphic_4
megamorphic_5
megamorphic_6
3.8529128184618315 ns/op
4.43550921376209
Graal JIT
3.0746344887370856 ns/op
3.553891709351798 ns/op
Graal JIT
4.082841769562822 ns/op
3.4070139557437713 ns/op
C2 JIT
4.008922983807572 ns/op
3.2719081770539558 ns/op
C2 JIT
3.769579570199055 ns/op
3.223400196836296 ns/op
C2 JIT
3.6216373543247484 ns/op
3.2714312594071577 ns/op
0
1
2
3
4
3:c1_c2
3:graal

bimorphic()
<<under the hood>>

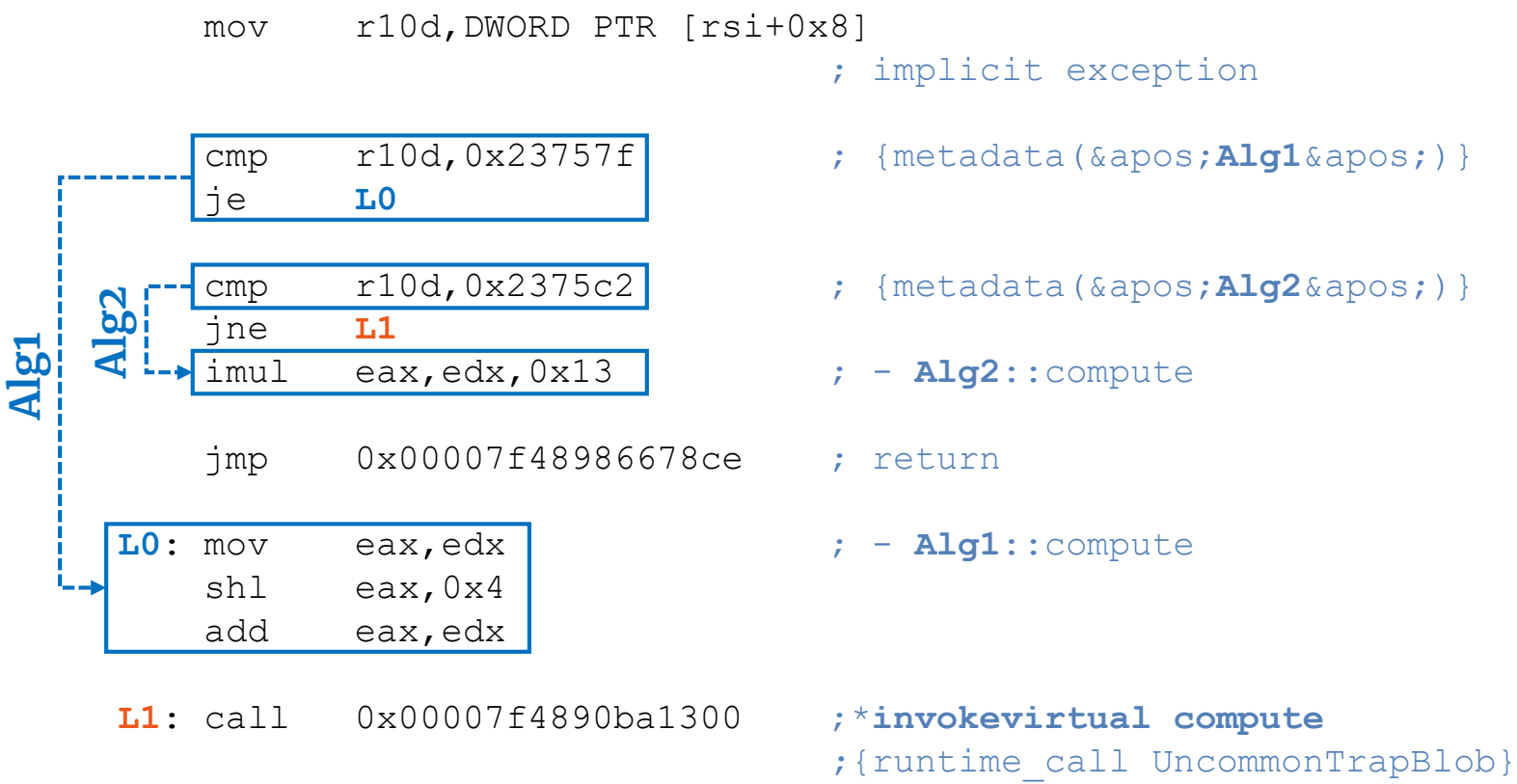
C2 JIT, level 4, execute() ; generated assembly - bimorphic case

```
        mov    r10d,DWORD PTR [rsi+0x8]
                                        ; implicit exception

        cmp    r10d,0x23757f            ; {metadata(&apos;Alg1&apos;)}
        je     L0

        cmp    r10d,0x2375c2            ; {metadata(&apos;Alg2&apos;)}
        jne    L1
        imul   eax,edx,0x13             ; - Alg2::compute

        jmp    0x00007f48986678ce       ; return

   L0:  mov    eax,edx                  ; - Alg1::compute
        shl    eax,0x4
        add    eax,edx

   L1:  call   0x00007f4890ba1300       ;*invokevirtual compute
                                        ;{runtime_call UncommonTrapBlob}
```

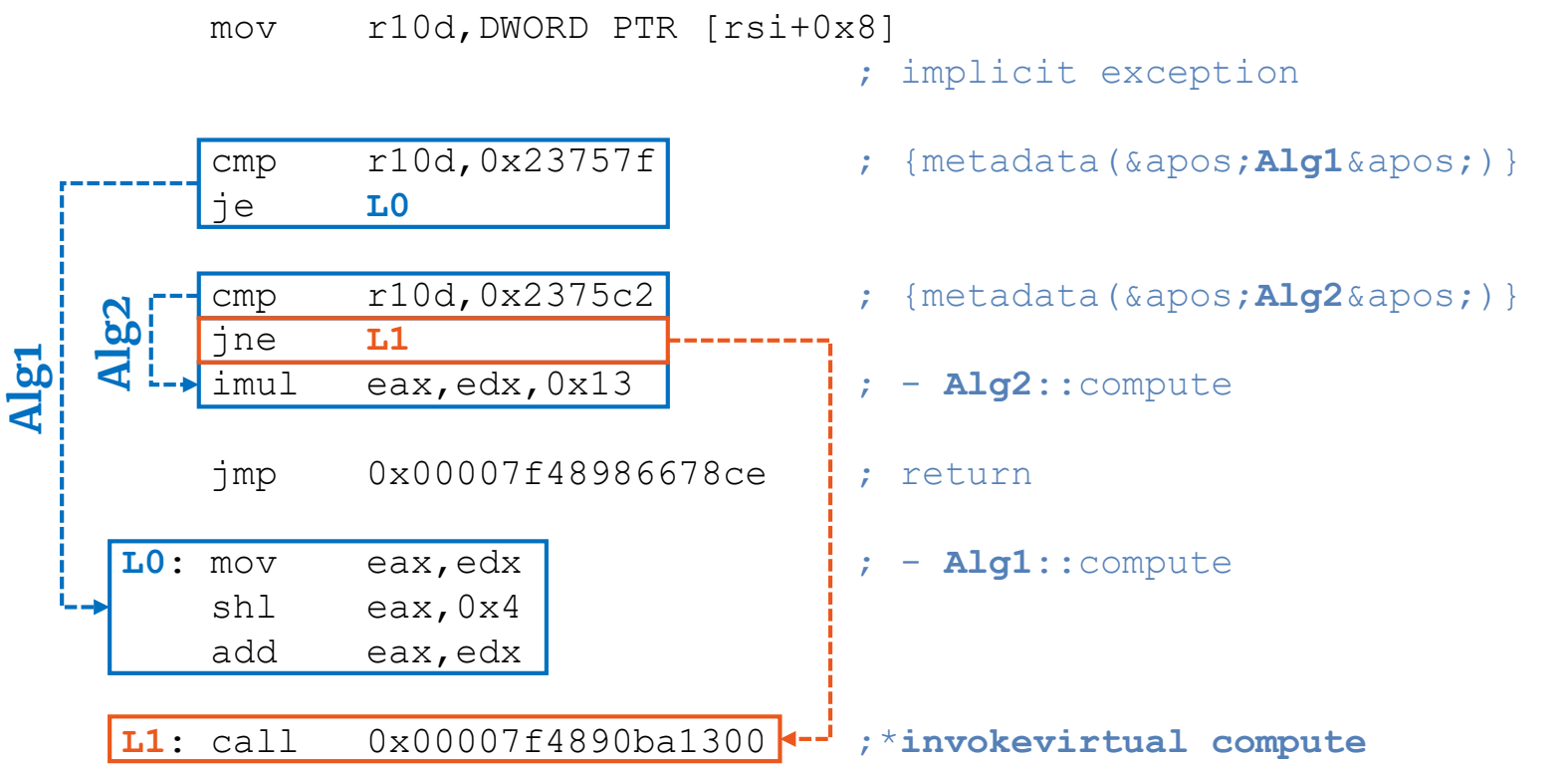
C2 JIT, level 4, execute() ; generated assembly - bimorphic case



C2 JIT, level 4, execute() ; generated assembly - bimorphic case



C2 JIT, level 4, execute() ; generated assembly - bimorphic case



Graal JIT, execute() ; generated assembly - bimorphic case

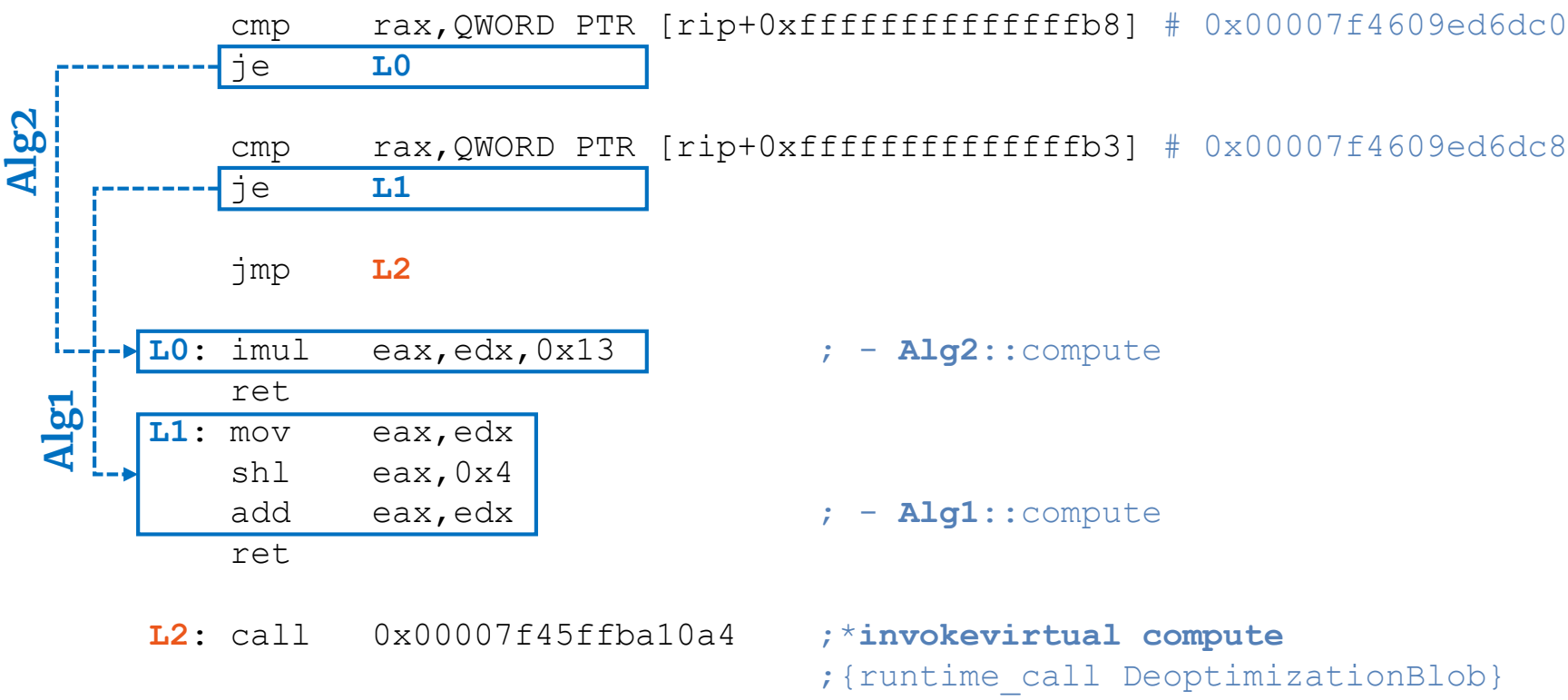
```
        cmp    rax,QWORD PTR [rip+0xfffffffffffffb8] # 0x00007f4609ed6dc0
        je     L0

        cmp    rax,QWORD PTR [rip+0xfffffffffffffb3] # 0x00007f4609ed6dc8
        je     L1

        jmp    L2

   L0:  imul   eax,edx,0x13          ; - Alg2::compute
        ret
   L1:  mov    eax,edx
        shl    eax,0x4
        add    eax,edx               ; - Alg1::compute
        ret

   L2:  call   0x00007f45ffba10a4    ;*invokevirtual compute
                                     ;{runtime_call DeoptimizationBlob}
```

Graal JIT, execute() ; generated assembly - bimorphic case

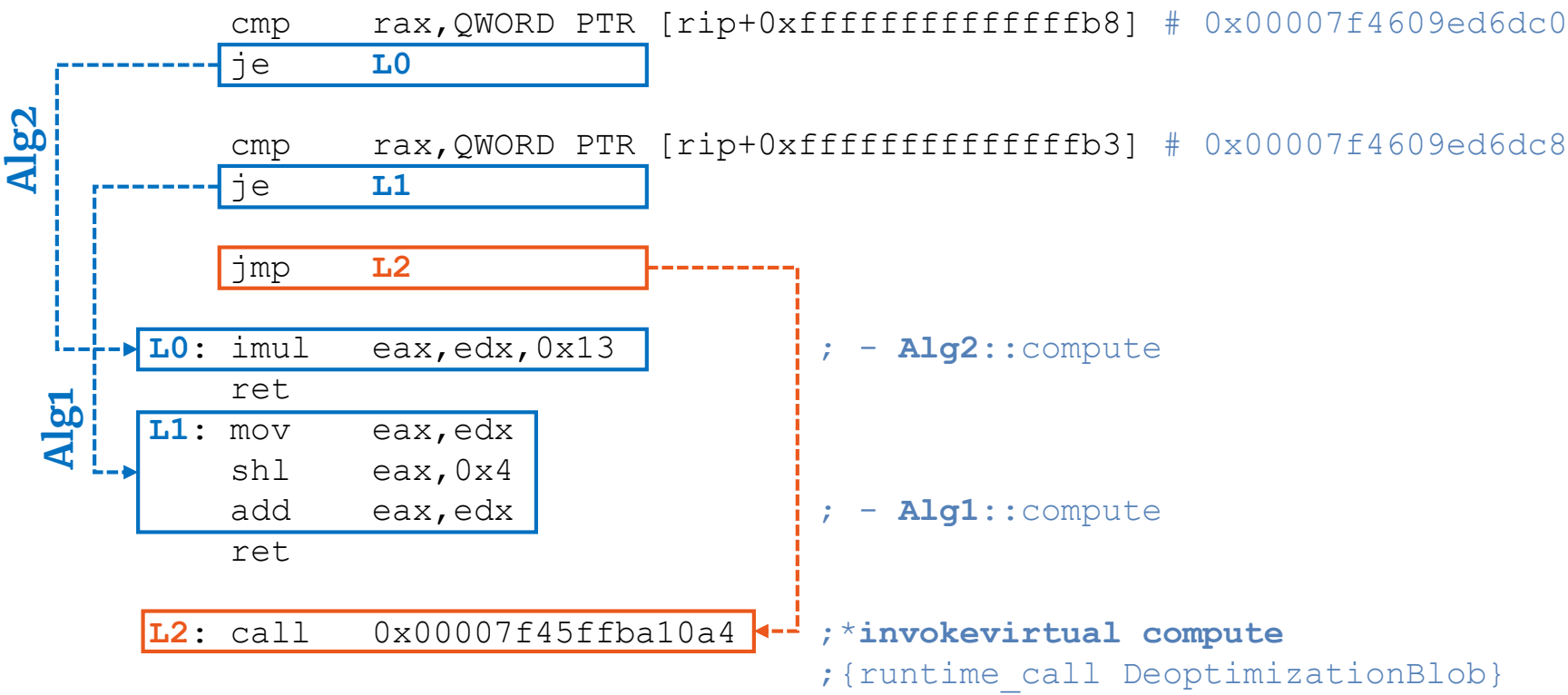
```
        cmp    rax,QWORD PTR [rip+0xfffffffffffffff8] # 0x00007f4609ed6dc0
        je     L0

        cmp    rax,QWORD PTR [rip+0xfffffffffffffff3] # 0x00007f4609ed6dc8
        je     L1

        jmp    L2

L0:     imul   eax,edx,0x13              ; - Alg2::compute
        ret
L1:     mov    eax,edx
        shl    eax,0x4
        add    eax,edx                   ; - Alg1::compute
        ret

L2:     call   0x00007f45ffba10a4        ;*invokevirtual compute
                                         ;{runtime_call DeoptimizationBlob}
```

Alg1

Graal JIT, execute() ; generated assembly - bimorphic case



Graal JIT, execute() ; generated assembly - bimorphic case



Conclusions - bimorphic call site

Both compilers (**Graal JIT** and **C2 JIT**) optimize bimorphic call sites in a similar approach:

- inline caching plus several comparison/jump checks
- adds a deoptimization routine (Graal JIT) or an uncommon trap (C2 JIT) for another, unknown, possible target invocation

megamorphic_3()

<<under the hood>>

C2 JIT, level 4, execute() ; generated assembly - megamorphic_3 case

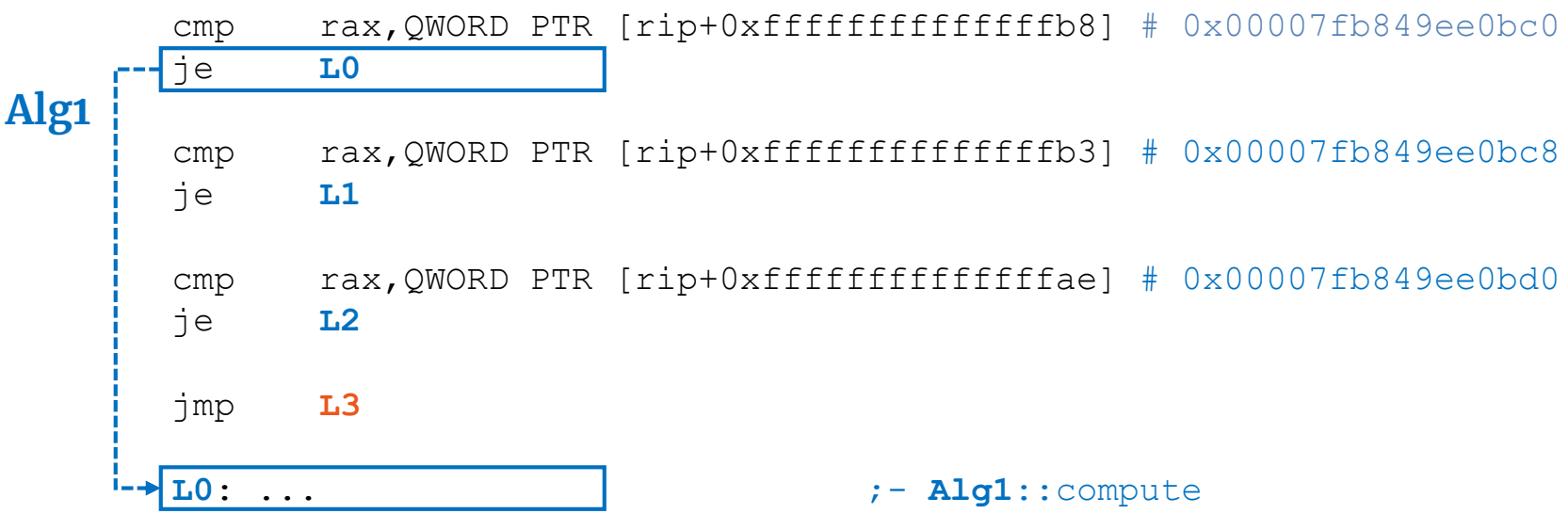
```
movabs rax,0xffffffffffffffff
call   0x00007fd65cb9fb80          ;*invokevirtual compute
                                   ;{virtual_call}
```

Not any further optimization.

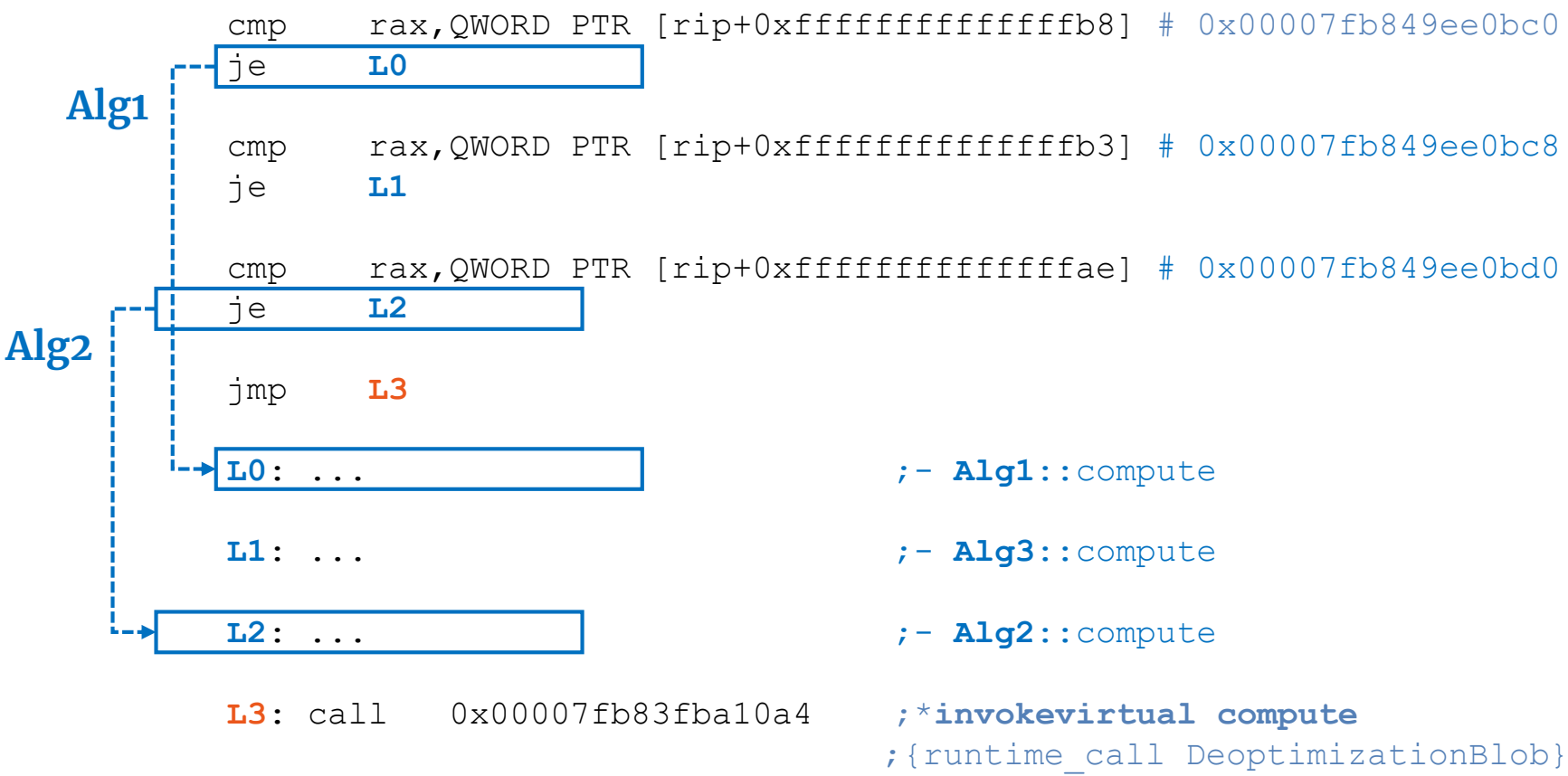
Graal JIT, execute() ; generated assembly - megamorphic_3 case

```
cmp    rax,QWORD PTR [rip+0xfffffffffffffb8] # 0x00007fb849ee0bc0
je     L0

cmp    rax,QWORD PTR [rip+0xfffffffffffffb3] # 0x00007fb849ee0bc8
je     L1

cmp    rax,QWORD PTR [rip+0xfffffffffffffae] # 0x00007fb849ee0bd0
je     L2

jmp    L3

L0: ...                                  ;- Alg1::compute

L1: ...                                  ;- Alg3::compute

L2: ...                                  ;- Alg2::compute

L3: call   0x00007fb83fba10a4    ;*invokevirtual compute
                                 ;{runtime_call DeoptimizationBlob}
```

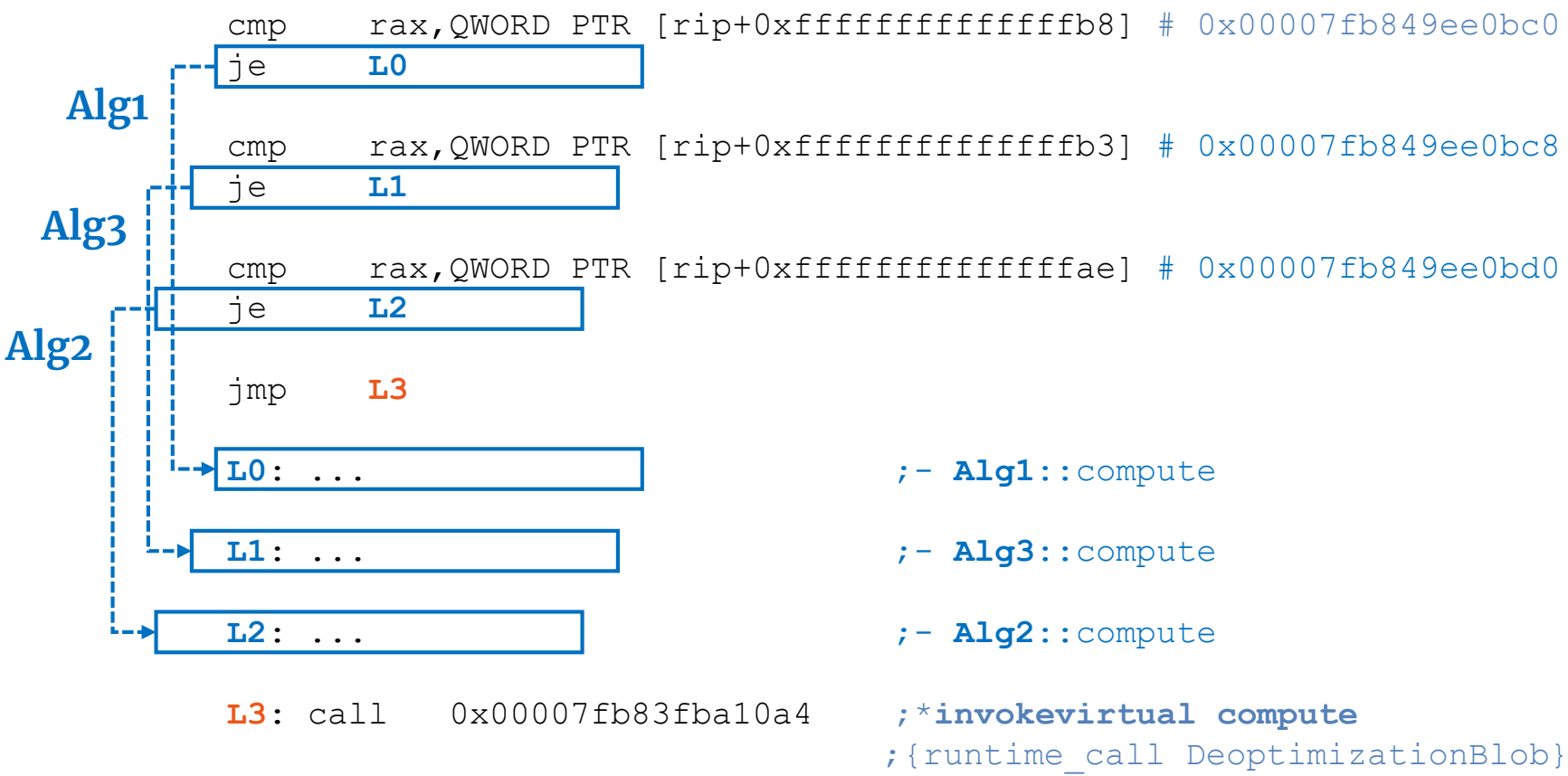
Graal JIT, execute() ; generated assembly - megamorphic_3 case



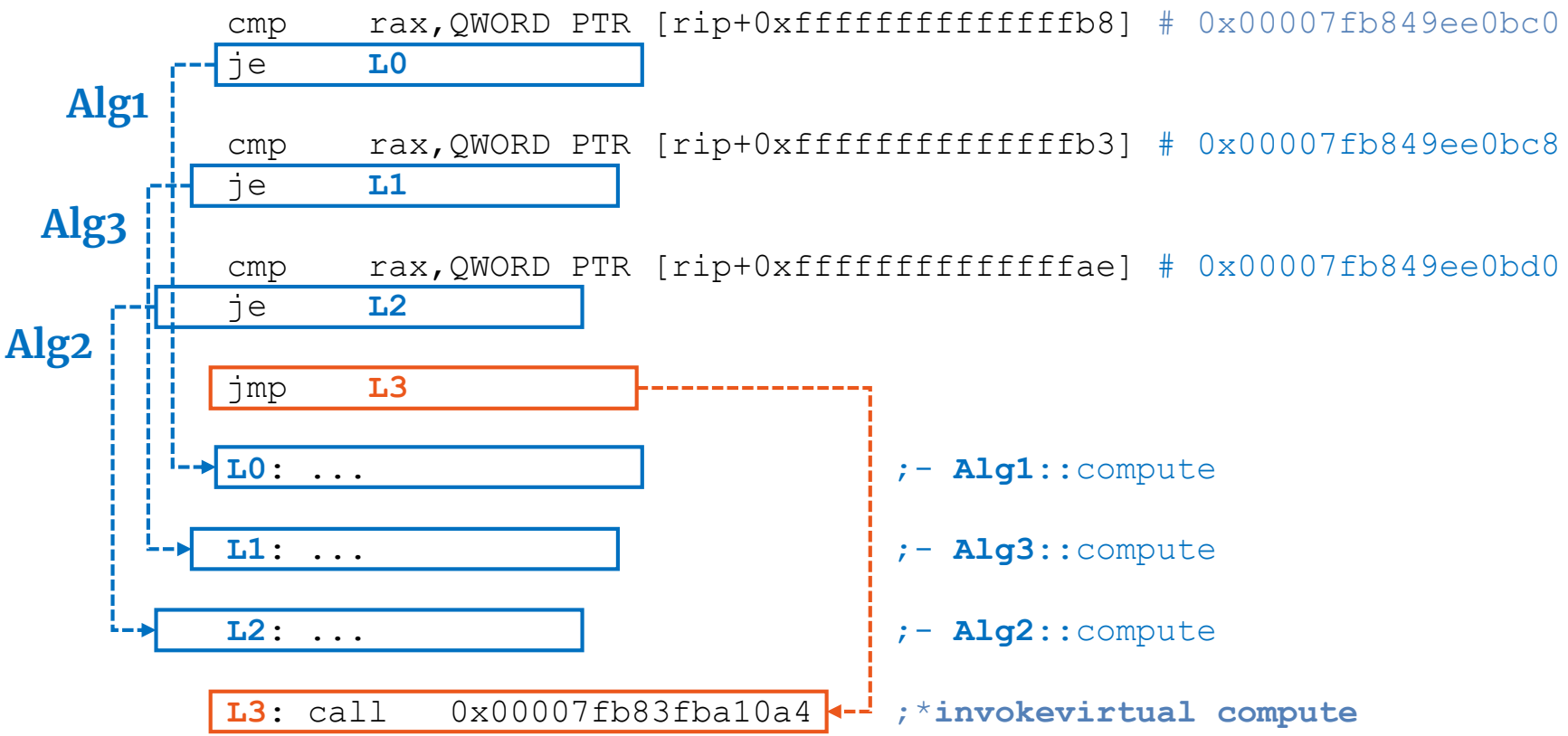
Graal JIT, execute() ; generated assembly - megamorphic_3 case



Graal JIT, execute() ; generated assembly - megamorphic_3 case



Graal JIT, execute() ; generated assembly - megamorphic_3 case



Conclusions - megamorphic_3 call site

Graal JIT optimizes three target method calls as follows:

- inline caching plus several comparison/jump checks
- adds a deoptimization routine for another (unknown) possible target invocation.

Conclusions - megamorphic_3 call site

Graal JIT optimizes three target method calls as follows:
- inline caching plus several comparison/jump checks
- adds a deoptimization routine for another (unknown) possible target invocation.

C2 JIT does not perform any further optimization, there is a virtual call (e.g. vtable lookup) → due to "polluted profile" context (i.e. method is used in many different contexts with independent operand types).

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:	tbd
Component/s:	hotspot		
Labels:	c1 performance tiered-compilation		
Subcomponent:	compiler		

People

Assignee:	Unassigned
Reporter:	John Rose
Votes:	0 Vote for this issue
Watchers:	4 Start watching this issue

Dates

Created:	2013-05-24 19:55
Updated:	2018-10-04 23:10

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.

Background:

The interpreter collects type profiles for invoke receivers and type checks. The profiles are crucial to later optimizing compilation. Tier one currently emulates the interpreter with respect to profiling.

Conclusions - megamorphic_3 call site

Graal JIT optimizes three target method calls as follows:
- inline caching plus several comparison/jump checks
- adds a deoptimization routine for another (unknown) possible target invocation.

C2 JIT does not perform any further optimization, there is a virtual call (e.g. vtable lookup) → due to "polluted profile" context (i.e. method is used in many different contexts with independent operand types).

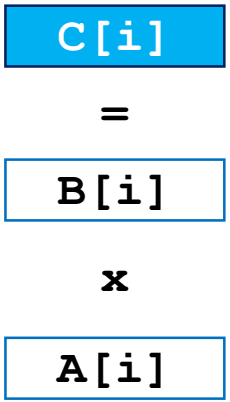
Note: megamorphic_4, megamorphic_5, and megamorphic_6 are optimized in similar ways by Graal JIT and C2 JIT

Vectorization & Loop Optimizations

04

Scenario: `C[i] = A[i] x B[i]`

Scalar version processes a single pair of operands at a time.



Scenario: `C[i] = A[i] x B[i]`

Scalar version processes a single pair of operands at a time.

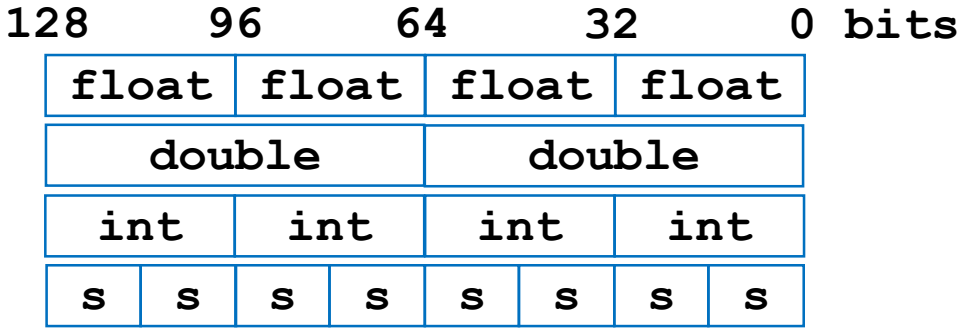
Vectorized version carries out the same instructions on multiple pairs of operands at once.

C[i]	C[i+1]	C[i+2]	C[i+3]
=	=	=	=
B[i]	B[i+1]	B[i+2]	B[i+3]
x	x	x	x
A[i]	A[i+1]	A[i+2]	A[i+3]

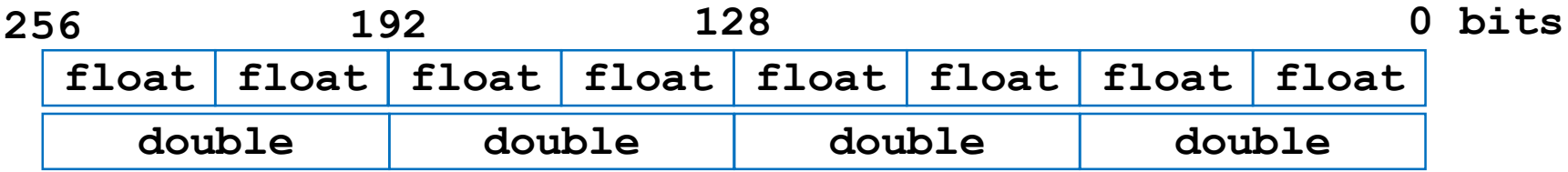
Vector operations implemented via **SIMD** (Single Instruction Multiple Data) ops.

x86 SIMD extensions

XMM registers



YMM registers



Note: few registers mentioned in current presentation. Check Intel manuals for a complete list.

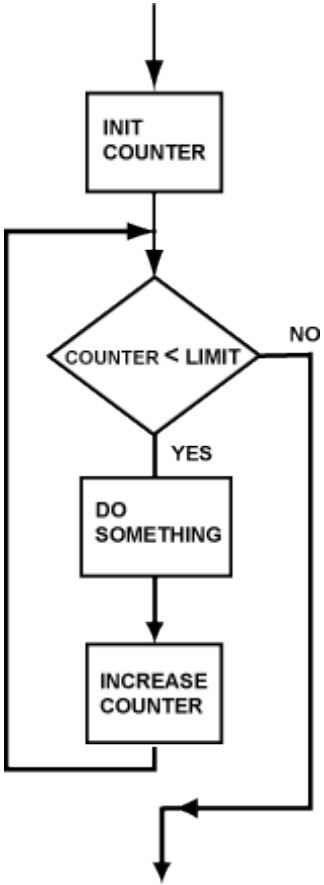
Modern compilers analyze loops in serial code to identify if they could be vectorized (i.e. perform loop transformations) - **auto vectorization**

Loops requirements for auto vectorization:

- countable loops → only unrolled loops
- single entry and exit
- straight-line code (e.g. no switch, if with masking is allowed)
- only for most inner loop (caution in case of loop interchange or loop collapsing)
- no functions calls, only intrinsics are allowed

Countable Loops

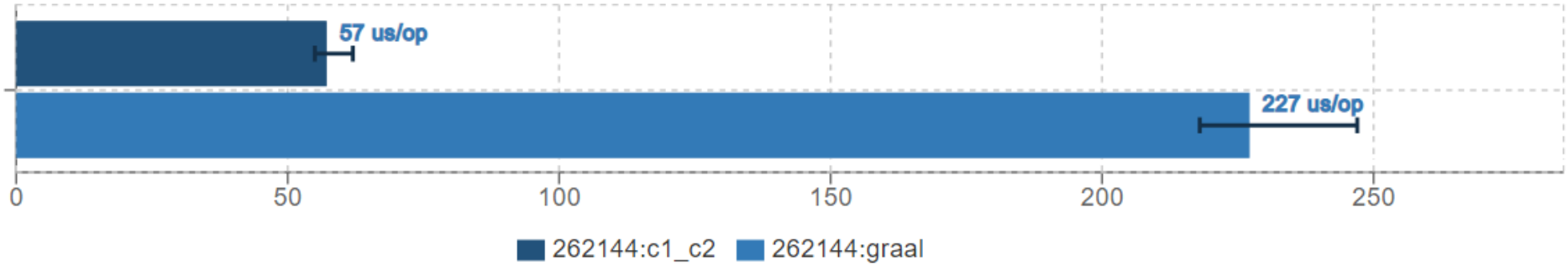
```
1) for (int i = start; i < limit; i += stride) {
       // loop body
   }

2) int i = start;
   while (i < limit) {
       // loop body
       i += stride;
   }
```

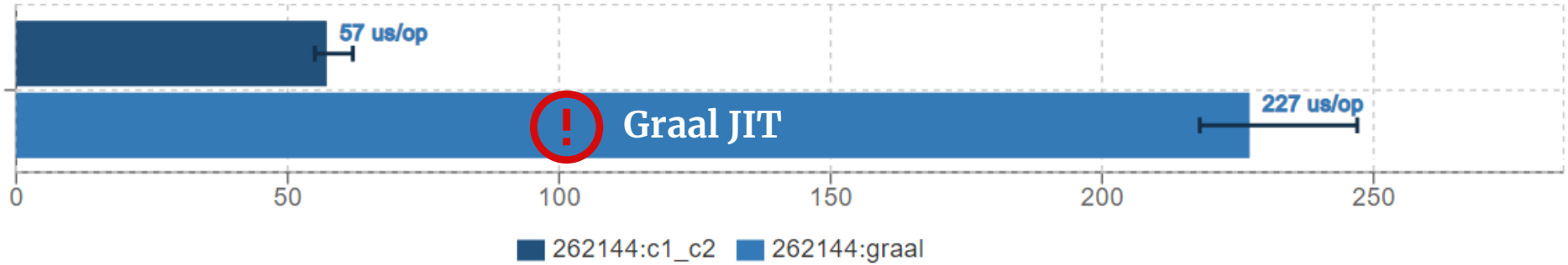


- limit is loop invariant
- stride is constant (compile-time known)

```
C[i] = A[i] x B[i]
```

```
@Param({"262144"})
private int size;
private float[] A, B, C;

@Benchmark
public float[] multiply_2_arrays_elements() {
    for (int i = 0; i < size; i++) {
        C[i] = A[i] * B[i];
    }
    return C;
}
```


57 us/op
227 us/op
0
50
100
150
200
250
262144:c1_c2
262144:graal


57 us/op
227 us/op
Graal JIT
0
50
100
150
200
250
262144:c1_c2
262144:graal

C2 JIT loop vectorization pattern

Scalar pre-loop

alignment (CPU caches)

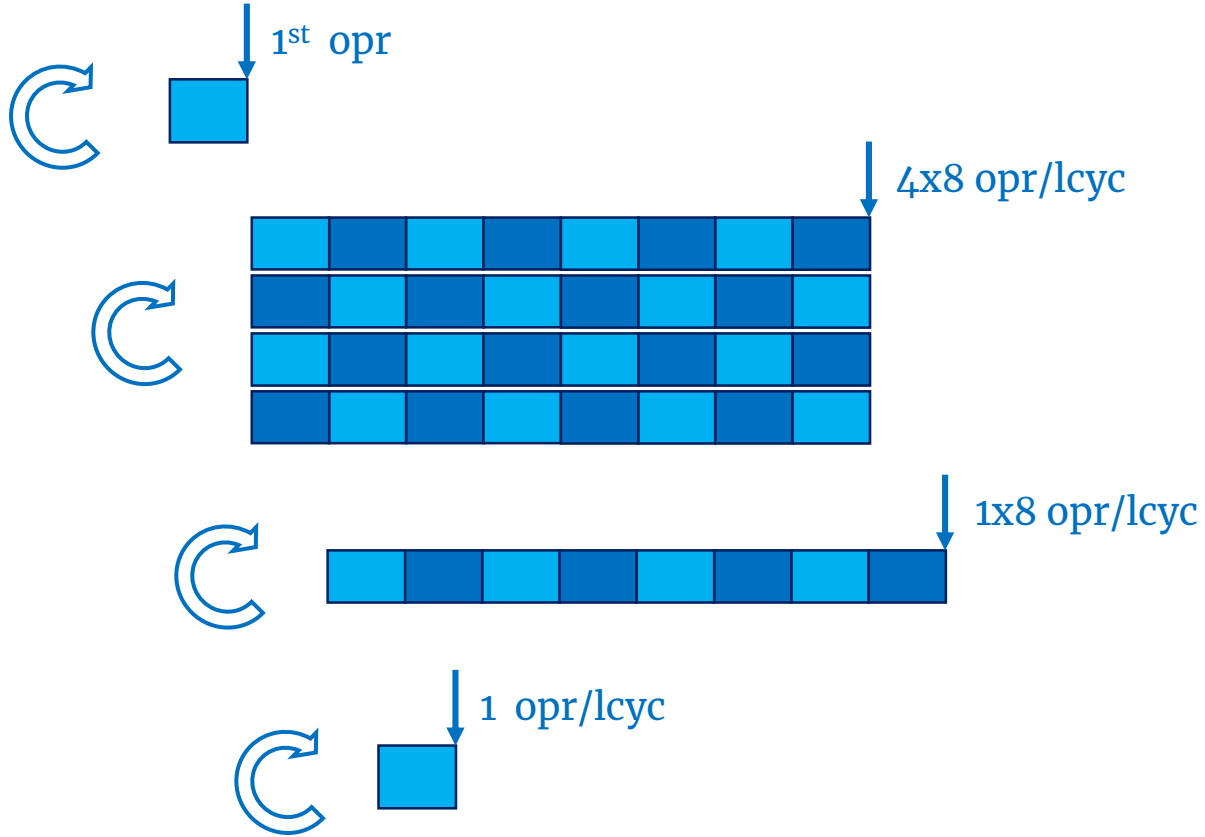
Vectorized main-loop

e.g. YMM registers

Vectorized post-loop

e.g. YMM registers

Scalar post-loop



Legend: opr → loop body (i.e. operand(s) / array element(s)); lcyc → loop cycle

C2 JIT, Scalar pre-loop (1 float operation/loop cycle)

```
L0000:
    vmovss   xmm2,DWORD PTR [rdi+r13*4+0x10]          ;*faload
    vmulss   xmm2,xmm2,DWORD PTR [rsi+r13*4+0x10]     ;*fmul
    vmovss   DWORD PTR [rax+r13*4+0x10],xmm2          ;*fastore
    inc      r13d                                     ;*iinc
    cmp      r13d,ebx
jl L0000                                              ;*if_icmpge
```

```
/* Pattern: C[i] = A[i] x B[i]; i+=1 */
```

C2 JIT, Vectorized main-loop (4x8 float operations/loop cycle)

```
L0001:
    vmovdqu ymm2,YMMWORD PTR [rdi+r13*4+0x10]
    vmulps  ymm2,ymm2,YMMWORD PTR [rsi+r13*4+0x10]
    vmovdqu YMMWORD PTR [rax+r13*4+0x10],ymm2
    movsxd  rcx,r13d
    ...
    vmovdqu ymm2,YMMWORD PTR [rdi+rcx*4+0x70]
    vmulps  ymm2,ymm2,YMMWORD PTR [rsi+rcx*4+0x70]
    vmovdqu YMMWORD PTR [rax+rcx*4+0x70],ymm2        ;*fastore
    add     r13d,0x20                                ;*iinc
    cmp     r13d,r9d
jl L0001                                             ;*if_icmpge

/* Pattern: C[i:i+7] = A[i:i+7] x B[i:i+7]; i+=32 */
```

C2 JIT, Vectorized post-loop (8 float operations/loop cycle)

```
L0002:
    vmovdqu ymm2,YMMWORD PTR [rdi+r13*4+0x10]
    vmulps  ymm2,ymm2,YMMWORD PTR [rsi+r13*4+0x10]
    vmovdqu YMMWORD PTR [rax+r13*4+0x10],ymm2        ;*fastore
    add     r13d,0x8                                 ;*iinc
    cmp     r13d,r9d
jl L0002                                             ;*if_icmpge
```

```
/* Pattern: C[i:i+7] = A[i:i+7] x B[i:i+7]; i+=8 */
```

C2 JIT, Scalar post-loop (1 float operation/loop cycle)

```
L0004:
    vmovss   xmm4,DWORD PTR [rdi+r13*4+0x10]          ;*faload
    vmulss   xmm4,xmm4,DWORD PTR [rsi+r13*4+0x10]     ;*fmul
    vmovss   DWORD PTR [rax+r13*4+0x10],xmm4          ;*fastore
    inc      r13d                                     ;*iinc
    cmp      r13d,r11d
jl L0004                                              ;*iload_1
```

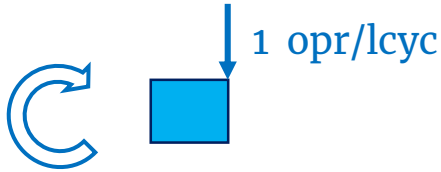
```
/* Pattern: C[i] = A[i] x B[i]; i+=1 */
```

Graal JIT loop optimization pattern

Loop peeling

alignment (CPU caches)

Scalar loop



Lack of **loop vectorization** support.
No **loop unrolling**.

Legend: opr → loop body (i.e. operand(s) / array element(s)); lcyc → loop cycle

Graal JIT, Loop peeling (2 float operations)

```
shl     rcx,0x3                                  ;*getfield B
vmovss  xmm0,DWORD PTR [rcx+r13*4+0x10]          ;*faload
shl     r8,0x3                                   ;*getfield A
vmulss  xmm0,xmm0,DWORD PTR [r8+r13*4+0x10]      ;*fmul
vmovss  DWORD PTR [r11+r13*4+0x10],xmm0          ;*fastore
mov     edx,r13d
inc     edx                                      ;*iinc

vmovss  xmm0,DWORD PTR [rcx+rdx*4+0x10]          ;*faload
vmulss  xmm0,xmm0,DWORD PTR [r8+rdx*4+0x10]      ;*fmul
vmovss  DWORD PTR [r11+rdx*4+0x10],xmm0          ;*fastore

lea     edx,[r13+0x2]                            ;*iinc
jmp L0001                                        ;main_loop
```

Graal JIT, Scalar loop (1 float operation/loop cycle)

```
L0000:
    vmovss xmm0,DWORD PTR [rcx+rdx*4+0x10]          ;*faload
    vmulss xmm0,xmm0,DWORD PTR [r8+rdx*4+0x10]      ;*fmul
    vmovss DWORD PTR [r11+rdx*4+0x10],xmm0          ;*fastore
    inc    edx                                      ;*iinc

L0001:
    cmp    r10d,edx
jg L0000                                            ;*if_icmpge
```

```
/* Pattern: C[i] = A[i] x B[i]; i+=1 */
```

Conclusions – multiply_2_arrays_elements

C2 JIT uses vectorization for:
- main loop - YMM AVX2 registers → 4x8 float operations/loop cycle
- post loop - YMM AVX2 registers → 8 float operations/loop cycle

C2 JIT handles the remaining post loop without unrolling and without vectorization, just one by one until reaches the end

Conclusions – multiply_2_arrays_elements

C2 JIT uses vectorization for:
- main loop - YMM AVX2 registers → 4x8 float operations/loop cycle
- post loop - YMM AVX2 registers → 8 float operations/loop cycle

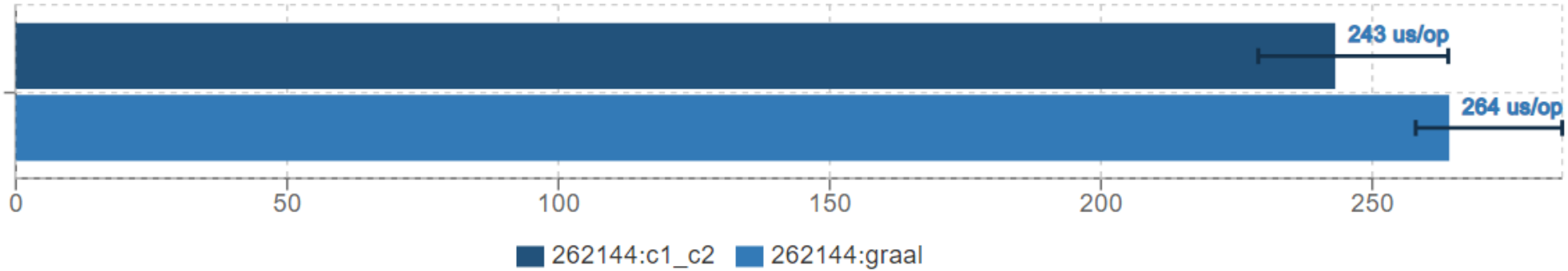
C2 JIT handles the remaining post loop without unrolling and without vectorization, just one by one until reaches the end

Graal JIT has no vectorization support → core feature missing in **OpenJDK**!

Graal JIT does not trigger loop unrolling (in this particular case)

C[l] = A[l] x B[l]
<<stride:long>>

```
@Param({"262144"})
private int size;
private float[] A, B, C;

@Benchmark
public float[] multiply_2_arrays_elements_long_stride() {
    for (long l = 0; l < size; l++) {
        C[(int) l] = A[(int) l] * B[(int) l];
    }
    return C;
}
```

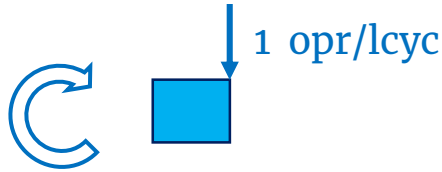


Note: the average timings for previous case (w/ **int stride**) were:

- **C2 JIT** - 57 us/op (i.e. ~4.2x faster)
- **Graal JIT** - 227 us/op (i.e. ~1.2x faster)

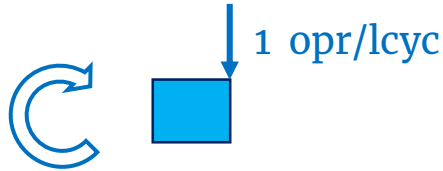
C2 JIT loop optimization pattern

Scalar loop



Graal JIT loop optimization pattern

Scalar loop



Lack of **loop vectorization** support.
No **loop unrolling**.

Legend: opr → loop body (i.e. operand(s) / array element(s)); lcyc → loop cycle

C2 JIT, Scalar loop (1 float operation/loop cycle)

```
L0000:
    vmovss DWORD PTR [rsi+rcx*4+0x10],xmm1          ;*goto
    mov    rcx,QWORD PTR [r15+0x108]
    add    r13,0x1                                  ;*ladd
    test   DWORD PTR [rcx],eax                      ;* SAFEPOINT POLL *
    cmp    r13,rdx
    jge    L0002                                    ;*ifge
    mov    ecx,r13d                                 ;*l2i
    vmovss xmm1,DWORD PTR [rax+rcx*4+0x10]          ;*faload
    vmulss xmm1,xmm1,DWORD PTR [r14+rcx*4+0x10]     ;*fmul
    cmp    ecx,r9d
jb L0000
```

```
/* Pattern: C[i] = A[i] x B[i]; i+=1 */
```

Graal JIT, Scalar loop (1 float operation/loop cycle)

```
L0000:
    mov    esi,ecx                                    ;*l2i
    vmovss xmm0,DWORD PTR [rdi+rsi*4+0x10]            ;*faload
    vmulss xmm0,xmm0,DWORD PTR [r8+rsi*4+0x10]        ;*fmul
    vmovss DWORD PTR [r11+rsi*4+0x10],xmm0            ;*fastore

    mov    rsi,QWORD PTR [r15+0x108]                  ;*lload_1
    test   DWORD PTR [rsi],eax                        ;* SAFEPOINT POLL *
    inc    rcx                                        ;*ladd
    cmp r10,rcx
jg L0000                                              ;*ifge


/* Pattern: C[i] = A[i] x B[i]; i+=1 */
```

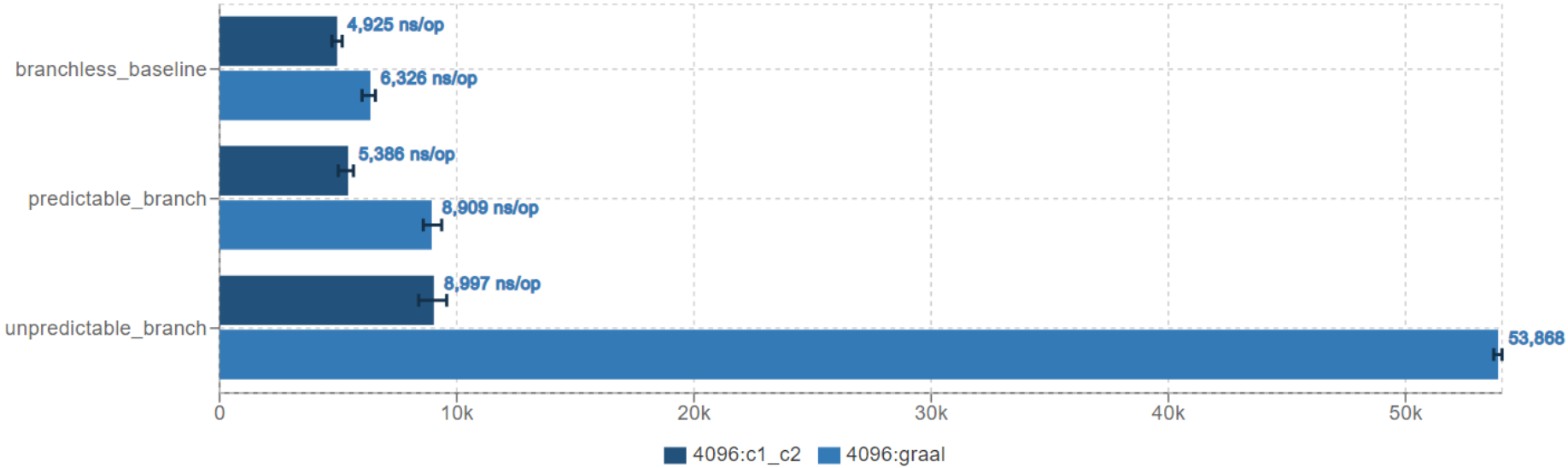
Conclusions – multiply_2_arrays_elements_long_stride

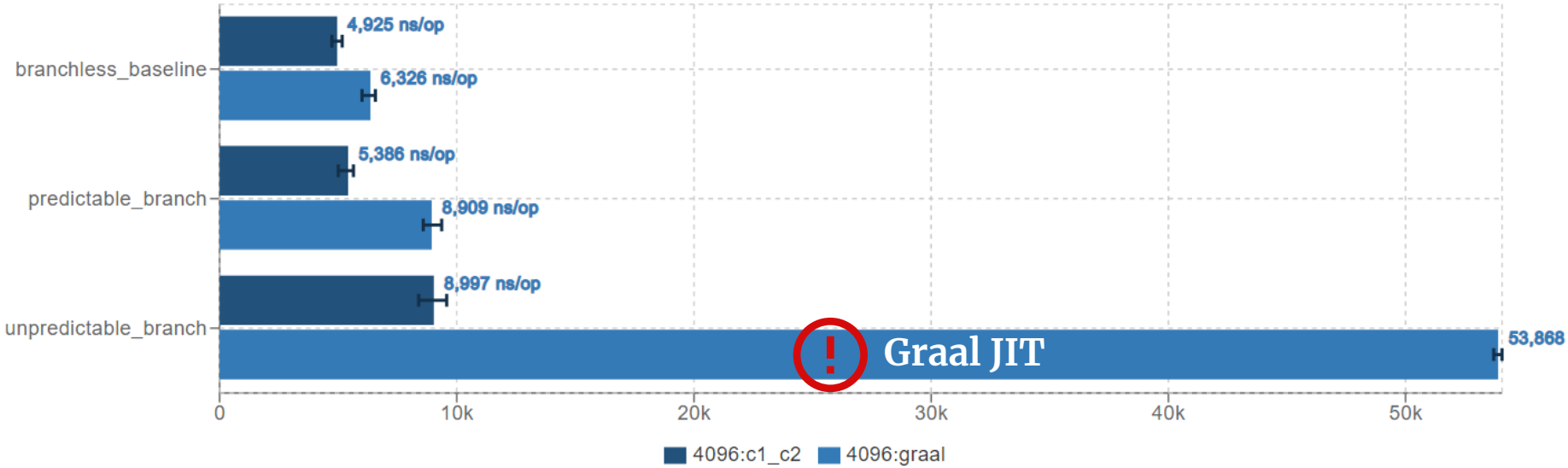
Both compilers (**Graal JIT** and **C2 JIT**) optimizes the float arrays elements multiplication with **long stride** in a similar fashion:

- one main scalar loop
- no loop unrolling
- no vectorization
- a safepoint poll is added within the loop

```
unpredictable_branch()
```

```
@Param({"4096"}) private int thresholdLimit;
private final int SIZE = 16_384;
private int[] array = new int[SIZE];

// Initialize array elements with values between [0, thresholdLimit)
// e.g. array[i] = random.nextInt(thresholdLimit); // i = [0, SIZE)

@Benchmark
public int unpredictable_branch() {
    int sum = 0;

    for (final int value : array) {
        if (value <= (thresholdLimit / 2)) {
            sum += value;
        }
    }
    return sum;
}
```


branchless_baseline
predictable_branch
unpredictable_branch
4,925 ns/op
6,326 ns/op
5,386 ns/op
8,909 ns/op
8,997 ns/op
53,868
0
10k
20k
30k
40k
50k
4096:c1_c2
4096:graal


branchless_baseline
predictable_branch
unpredictable_branch
4,925 ns/op
6,326 ns/op
5,386 ns/op
8,909 ns/op
8,997 ns/op
53,868
Graal JIT
0
10k
20k
30k
40k
50k
4096:c1_c2
4096:graal

C2 JIT loop optimization pattern

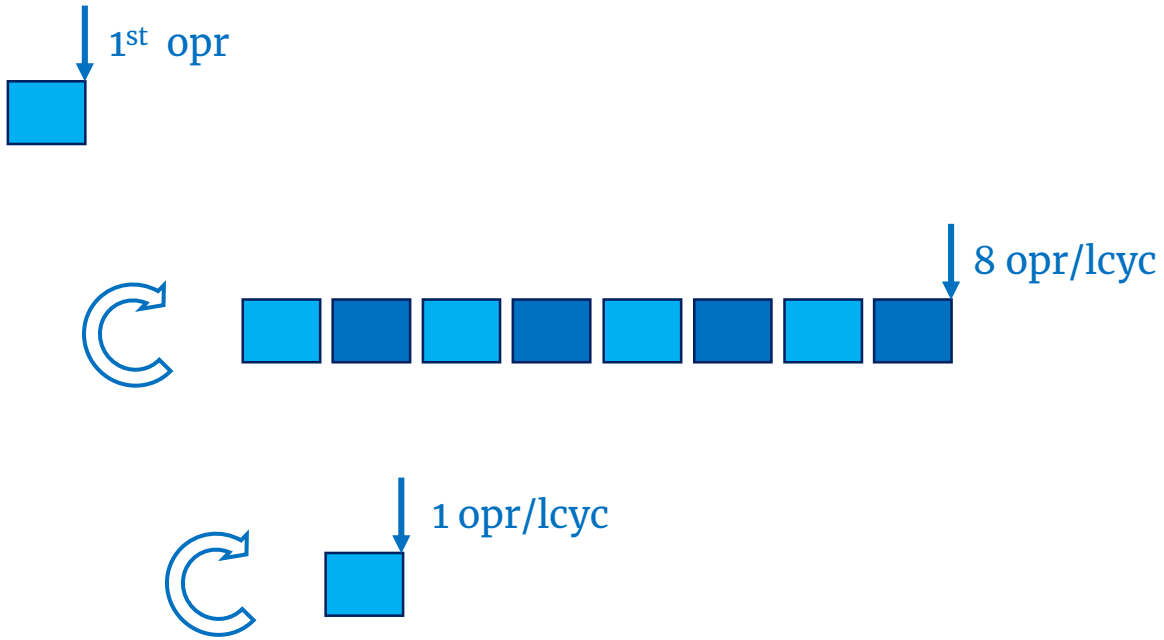
Loop peeling

alignment (CPU caches)

Scalar main-loop

e.g. unrolling factor = 8

Scalar post-loop



Legend: opr → loop body (i.e. operand(s) / array element(s)); lcyc → loop cycle

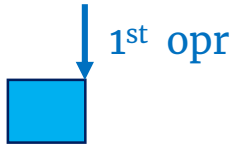
C2 JIT, Scalar main-loop (8 array elements/loop cycle)

```
L0000:
    mov    r10d,DWORD PTR [rbp+rdi*4+0x10]          ;*iaload
    mov    r9d,DWORD PTR [rbp+rdi*4+0x14]
    mov    r8d,DWORD PTR [rbp+rdi*4+0x18]
    mov    ebx,DWORD PTR [rbp+rdi*4+0x1c]
    mov    ecx,DWORD PTR [rbp+rdi*4+0x20]
    mov    edx,DWORD PTR [rbp+rdi*4+0x24]
    mov    esi,DWORD PTR [rbp+rdi*4+0x28]
    mov    r11d,eax
    add    r11d,r10d
    cmp    r10d,r13d                          ; r13d = thresholdLimit/2
    cmovle eax,r11d                           ;*iinc
    mov    r10d,DWORD PTR [rbp+rdi*4+0x2c]    ;*iaload
    ...
    add    edi,0x8                            ;*iinc
    cmp    edi,r14d
jl  L0000                                     ;*if_icmpge
```

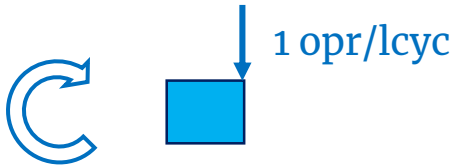
Graal JIT loop optimization pattern

Loop peeling

alignment (CPU caches)



Scalar loop



No **loop unrolling**.

Legend: opr → loop body (i.e. operand(s) / array element(s)); lcyc → loop cycle

Graal JIT, Scalar loop (1 array element/loop cycle)

```
L0000:
    mov     ecx,DWORD PTR [rax+r11*4+0x10]          ;*iaload
    cmp     ecx,r9d                                 ; r13d = thresholdLimit/2
    jg      L0003                                   ;*if_icmpgt
    add     r8d,ecx                                 ;*iadd
    inc     r11d                                    ;*iinc
    L0001: cmp r10d,r11d
jg L0000                                            ;*if_icmpge
```

Conclusions – unpredictable_branch

C2 JIT optimizes the loop as follows:
- main loop - with an unrolling factor of 8
- post loop - one by one, until it reaches the end

Conclusions – unpredictable_branch

C2 JIT optimizes the loop as follows:
- main loop - with an unrolling factor of 8
- post loop - one by one, until it reaches the end

Graal JIT does not unroll the loop (second example without unrolling!)

Summary & Takeaways

05

	Graal JIT	C2 JIT
Scalar Replacement	✓	
Virtual Calls	✓	
Vectorization	!	✓
Loop Unrolling		✓
Intrinsics		✓

Scalar Replacement: nice optimizations by Graal JIT (e.g. Partial Escape Analysis).
Virtual Calls: better optimized by Graal JIT (e.g. megamorphic call sites).
Vectorization: core compiler feature "missing" in Graal JIT (OpenJDK).
Loop Unrolling: not found in current Graal JIT test cases. Missing or just limited!?
Intrinsics: not covered here but the support is lower in Graal JIT. See reference.

Application Developer Guidelines

	Graal JIT	C2 JIT
a lot of objects created	✓	
high degree of polymorphic calls	✓	
myriad of tiny nested calls (i.e. functional programming)	✓	
a lot of computational loops / mathematical operations		✓
quicker JIT start-up performance / lower JIT runtime overhead		✓

Note: please take these guidelines with precaution, they are not general valid!
"Measure, do not guess" and apply what fits your application/context.

Open JDK support

Version	Graal JIT	C2 JIT
8	!	✓
9	!	✓
10	✓	✓
11	✓	✓
12	✓	✓
13	✓	✓

- for Java 8 there are Graal JIT backports available
- for Java 9 Graal JIT needs to be manually added on top of the OpenJDK distribution

Garbage Collectors compatibility in Open JDK

	Graal JIT	C2 JIT
Serial	✓	✓
Parallel/ParallelOld	✓	✓
CMS	!	✓
G1	✓	✓
Z	!	✓
Epsilon	!	✓
Shenandoah [1]	!	✓

[1] – included in RedHat OpenJDK builds

Note: JVMCI Compiler does not support selected GC: {CMS, Z; Shenandoah; Epsilon}


Thanks

Resources

Slides

IonutBalosin.com/talks

Further Readings

1. https://IonutBalosin.com/2019/04/jvm-jit-compilers-benchmarks-report-19-04
2. https://graalworkshop.github.io/2019/Performance-Characterization-And-Optimizations-In-Graal-At-Intel.pdf by Jean-Philippe Halimi

IonutBalosin.com/training

@IonutBalosin

ANNEX

Virtual Calls
<another approach>

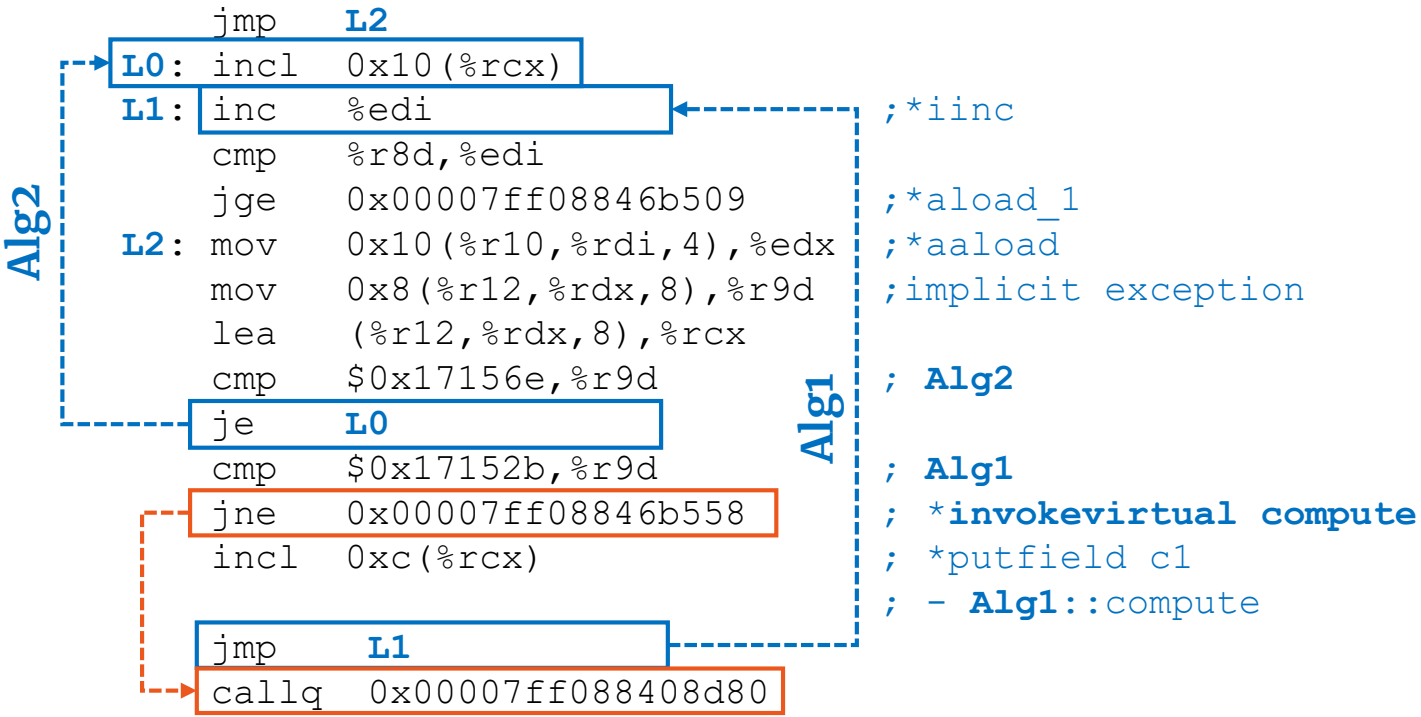
```
private CMath[] instances;

@Param("144000")
private int size;

// CMath = {Alg1, Alg2, Alg3, Alg4, Alg5, Alg6}
@Benchmark
@OperationsPerInvocation(144000)
public void test() {
    for (CMath instance : instances) {
        instance.compute();
    }
}
```

megamorphic call site

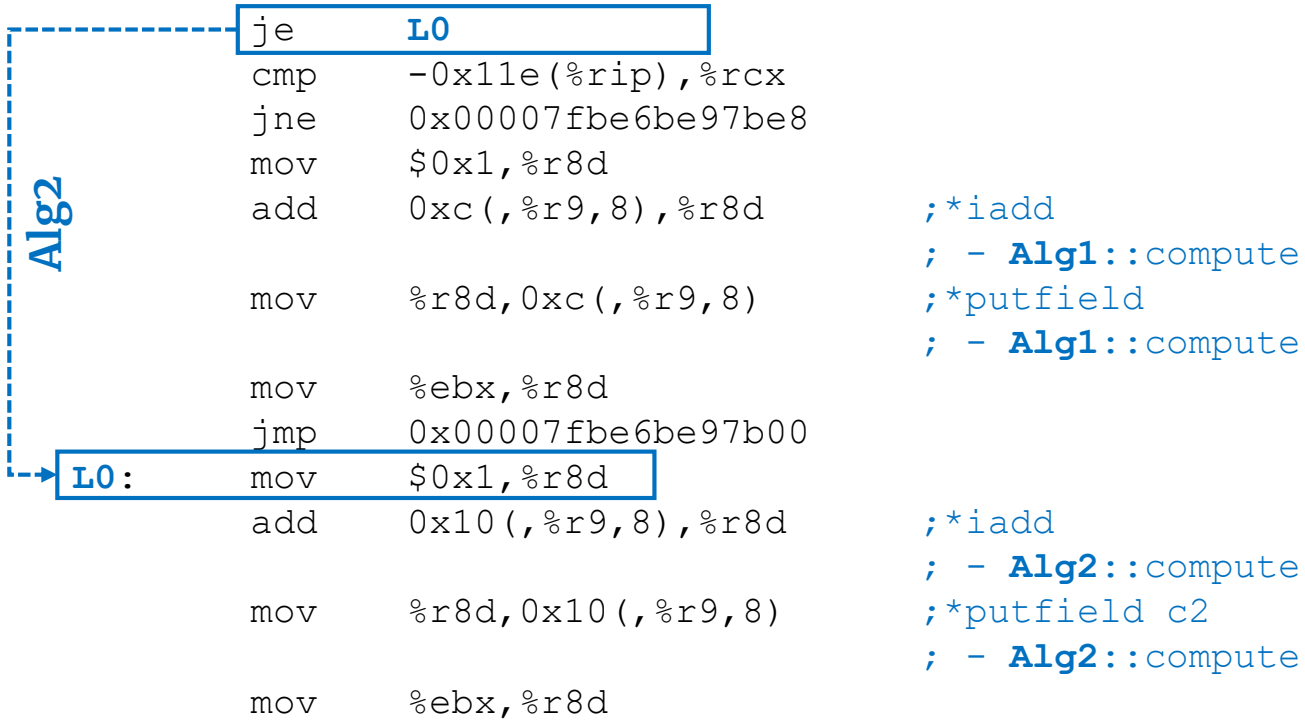
```
static abstract class CMath {
    int c1, c2, c3, c4, c5, c6;
    public abstract int compute();
}
static class Alg1 extends CMath {
    public int compute() {
        return ++c1;
    }
}
static class Alg2 extends CMath {
    public int compute() {
        return ++c2;
    }
}
// similar implementations for Alg3, Alg4, Alg5, Alg6
```

bimorphic()
<<under the hood>>

C2 JIT, level 4, execute() ; generated assembly - bimorphic case



Graal JIT, execute() ; generated assembly - bimorphic case



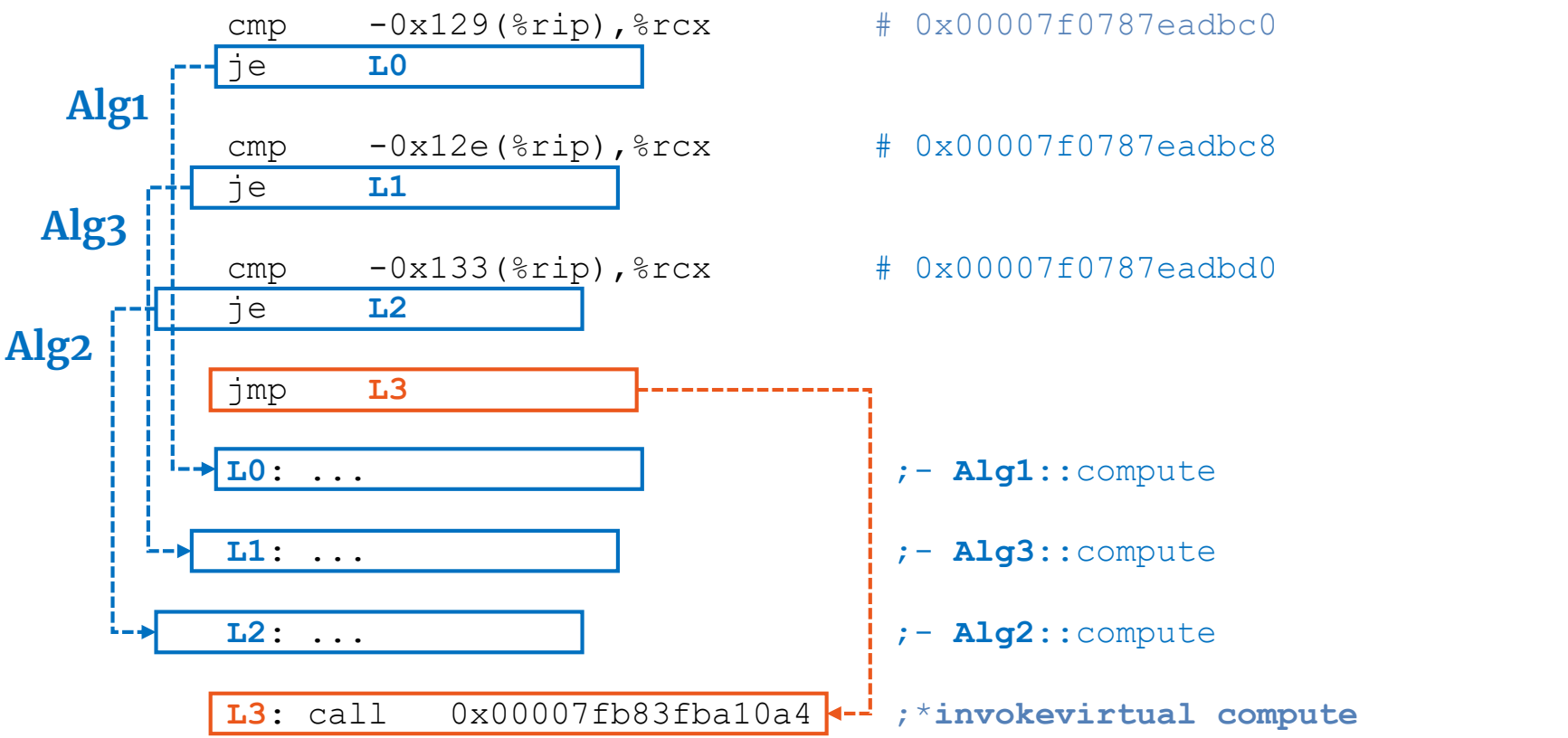
megamorphic_3()
<<under the hood>>

C2 JIT, level 4, execute() ; generated assembly - megamorphic_3 case

```
movabs rax,0xffffffffffffffff
call   0x00007fd82cab6700        ;*invokevirtual compute
                                 ;{virtual_call}
```

Not any further optimization.

Graal JIT, execute() ; generated assembly - megamorphic_3 case



```
unpredictable_branch()
<continued>
```

C2 JIT, Loop peeling (1st array element)

```
mov     ebp,DWORD PTR [rsi+0x14]                ;*getfield array
mov     r10d,DWORD PTR [r12+rbp*8+0xc]          ;*arraylength
mov     eax,DWORD PTR [r12+rbp*8+0x10]          ;*iaload
mov     ecx,DWORD PTR [rsi+0x10]                ;*getfield thresholdLimit
mov     edi,0x1                                 ;main loop start index
```

C2 JIT, Scalar post-loop (1 array element/loop cycle)

```
L0001:
    mov     r11d,DWORD PTR [rbp+rdi*4+0x10]          ;*iaload

    mov     r9d,eax
    add     r9d,r11d
    cmp     r11d,r13d                                ; r13d = thresholdLimit/2
    cmovle  eax,r9d

    inc     edi                                      ;*iinc
    cmp     edi,r10d
jl L0001                                             ;*if_icmpge
```

Graal JIT, Loop peeling (1st array element)

```
mov     r11d,DWORD PTR [rsi+0x10]                ;*getfield thresholdLimit
mov     r8d,DWORD PTR [rax*8+0x10]               ;*iaload
shl     rax,0x3                                  ;*getfield array
mov     r11d,0x1
```