Graal Compiler Optimizations On AArch64

Xiaohong Gong
Sep 2019

Outline

- Graal Overview
- Graal Optimization status
- What optimizations have we done for AArch64
- The future work

Graal Overview

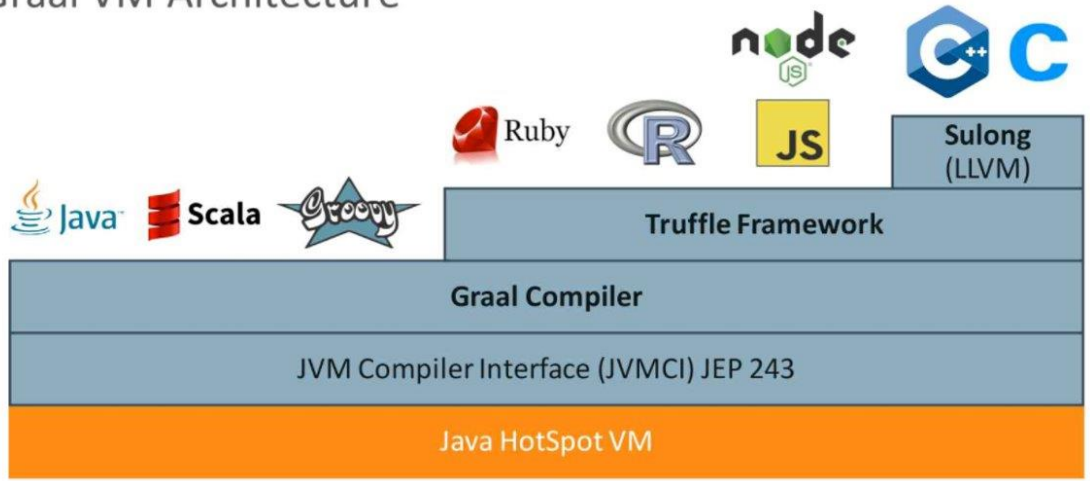
What is Graal?

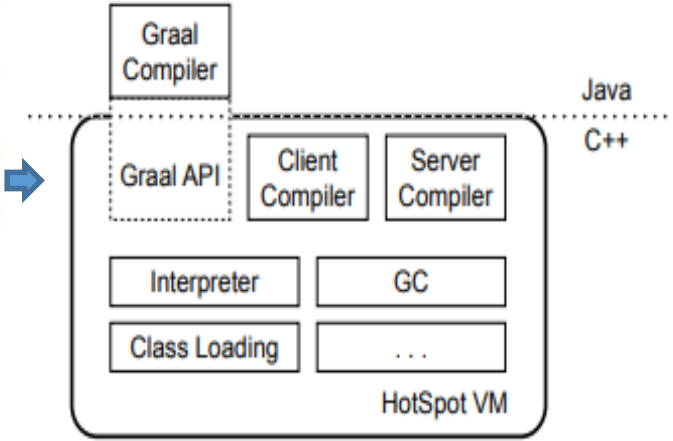
- A compiler written in Java, that compiles Java bytecode to machine codes (e.g. X86, AArch64, Sparc). It could be used in several ways:

Usage	Mode
Replacement for C2 compiler in Hotspot	JIT (Just-In-Time)
Hotspot AOT	AOT (Ahead-Of-Time)
Substrate VM	AOT

Note: Hotspot is the Virtual Machine (VM) in OpenJDK, and OpenJDK is the official Java implementation.

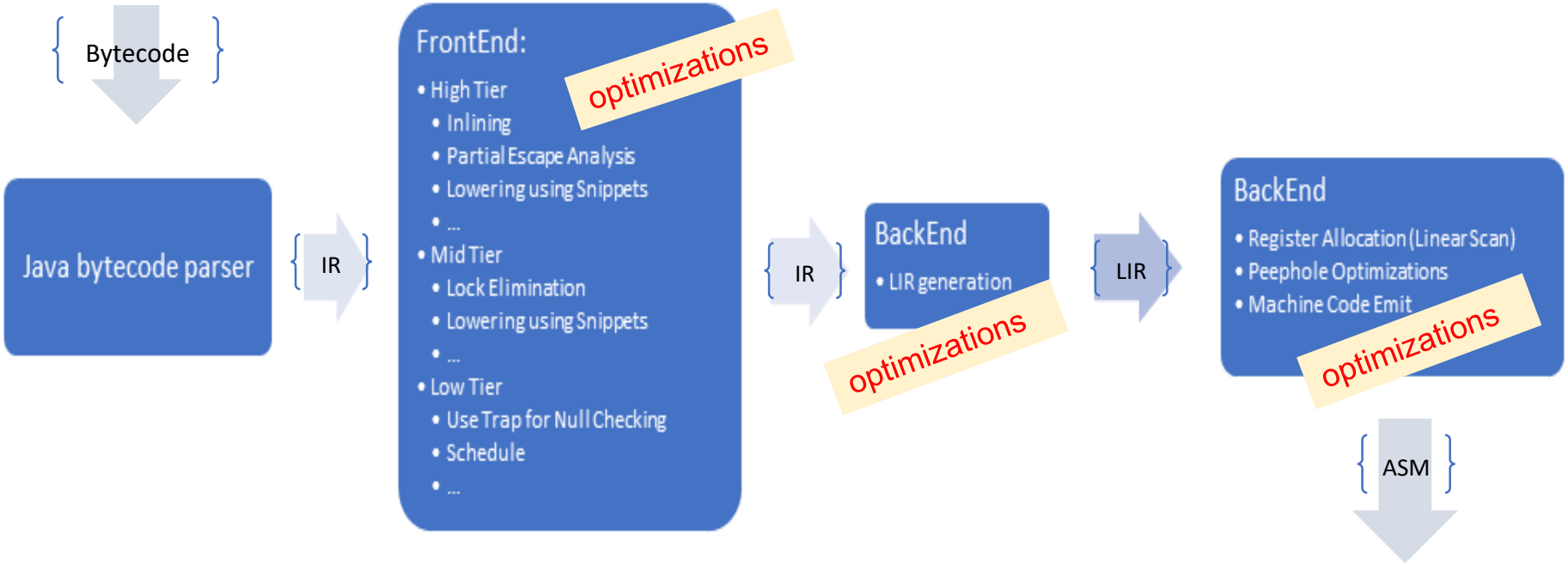
Graal VM Architecture





Note: Hotspot is the Virtual Machine (VM) in OpenJDK, and OpenJDK is the official Java implementation.

Compilation Workflow



Key Features

Written in Java:

- High development efficiency
- Safety
- Maintainable and extendable
- Good IDE support for debuggers

- Designed for speculative optimizations and deoptimization
- Designed for exact garbage collection (oop maps)
- Aggressive high-level optimizations (like PEA)
- Modular architecture (Compiler-VM separation)
- Compiling and optimizing itself is also a good optimization opportunity

Graal Optimization Status on AArch64

Graal vs C2

Mid-End Optimizations	Graal	C2
Inline, GVN, Constant propagation, Profile-guided Optimization(PGO), Canonicalization, Dead Code Elimination… (about 50 more)	Similar	Similar
Escape Analysis	Partial Escape Analysis	Non-partial EA
Lowering	Snippet	Non-snippet
Auto-vectorization	None	Superword-Level Parallelism (SLP) auto-vectorization
De-virtualization	Cache more virtual function calls	Cache fewer virtual function calls

Back-End Optimizations on AArch64	Graal	C2
Instruction Selection	Naïve instruction selection, no match rules on AArch64 until we added some	BURS(Bottom Up Rewrite System), match rules on AArch64
Intrinsics	Missing a lot	Most implemented
Instruction scheduling	Less tuned scheduler	Architecture specific scheduler
Register Allocation	Linear Scan	Graph coloring

Graal vs C2

Mid-End Optimizations	Graal	C2
Inline, GVN, Constant propagation, Profile-guided Optimization(PGO), Canonicalization, Dead Code Elimination… (about 50 more)	Similar	Similar
Escape Analysis	Partial Escape Analysis	Non-partial EA
Lowering	Snippet	Non-snippet
Auto-vectorization	None	Superword-Level Parallelism (SLP) auto-vectorization
De-virtualization	Cache more virtual function calls	Cache fewer virtual function calls

Back-End Optimizations on AArch64	Graal	C2
Instruction Selection	Naïve instruction selection, no match rules on AArch64 until we added some	BURS(Bottom Up Rewrite System), match rules on AArch64
Intrinsics	Missing a lot	Most implemented
Instruction scheduling	Less tuned scheduler	Architecture specific scheduler
Register Allocation	Linear Scan	Graph coloring

Graal vs C2

Mid-End Optimizations	Graal	C2
Inline, GVN, Constant propagation, Profile-guided Optimization(PGO), Canonicalization, Dead Code Elimination… (about 50 more)	Similar	Similar
Escape Analysis	Partial Escape Analysis	Non-partial EA
Lowering	Snippet	Non-snippet
Auto-vectorization	None	Superword-Level Parallelism (SLP) auto-vectorization
De-virtualization	Cache more virtual function calls	Cache fewer virtual function calls

Back-End Optimizations on AArch64	Graal	C2
Instruction Selection	Naïve instruction selection, no match rules on AArch64 until we added some	BURS(Bottom Up Rewrite System), match rules on AArch64
Intrinsics	Missing a lot	Most implemented
Instruction scheduling	Less tuned scheduler	Architecture specific scheduler
Register Allocation	Linear Scan	Graph coloring

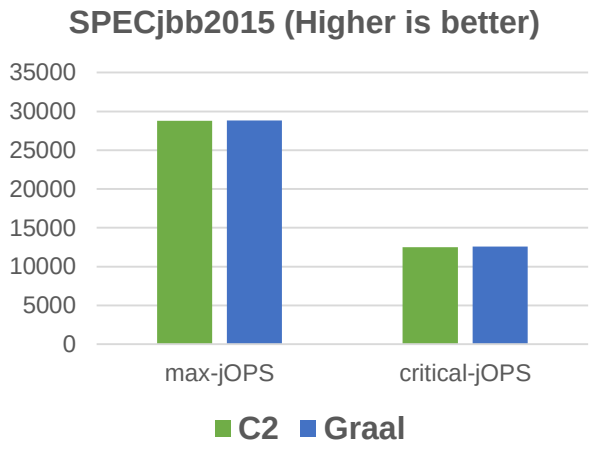
Graal vs C2

Mid-End Optimizations	Graal	C2
Inline, GVN, Constant propagation, Profile-guided Optimization(PGO), Canonicalization, Dead Code Elimination… (about 50 more)	Similar	Similar
Escape Analysis	Partial Escape Analysis	Non-partial EA
Lowering	Snippet	Non-snippet
Auto-vectorization	None	Superword-Level Parallelism (SLP) auto-vectorization
De-virtualization	Cache more virtual function calls	Cache fewer virtual function calls

Back-End Optimizations on AArch64	Graal	C2
Instruction Selection	Naïve instruction selection, no match rules on AArch64 until we added some	BURS(Bottom Up Rewrite System), match rules on AArch64
Intrinsics	Missing a lot	Most implemented
Instruction scheduling	Less tuned scheduler	Architecture specific scheduler
Register Allocation	Linear Scan	Graph coloring

Performance on AArch64

- SpecJbb2015 is a big benchmark developed for Java. It simulates a world-wide supermarket company's IT infrastructure.



Testing on AArch64 Cortex-A72 based platform with jdk11.

Conclusion

- Compared to C2, Graal is quite new. And even without much backend optimizations for AArch64 platform, it still provides similar performance to C2 for some cases.
- Graal is still missing some optimizations and new features of OpenJDK, and its performance for some cases is worse than C2. So we need to catch up with C2.
- With more optimizations added for AArch64, Graal's future performance will be compelling.

What have we done for optimizations on AArch64 ?

Optimizations we have done for AArch64

Back-End on AArch64:

- Instruction selections
  - Add match rules to delete or use light instructions like madd, ubfx/ubfm, mneg, tbz, etc.
  - Address reshaping and memory bulk zeroing for initialization.

  .....
- Intrinsics
  - Integer.bitCount/Long.bitCount
- Others
  - Memory barrier

Mid-End:

- Canonicalization
  - Integer div/rem optimization when the divisor is a constant.
  - If branch elimination.

If Elimination

Use conditional select(csel) instead of comparation and branch.

```
public static int generalMax(int m, int n) {
    if (m >= n) {
        return m;
    }
    return n;
}
```

code gen

```
0x0000ffff84638014: cmp w3, w2
0x0000ffff84638018: b.gt        0x0000ffff84638034
0x0000ffff8463801c: mov w0, w2
0x0000ffff84638020:
0x0000ffff84638024:
0x0000ffff84638028: ...
0x0000ffff8463802c:
0x0000ffff84638030:
0x0000ffff84638034: mov w0, w3
```

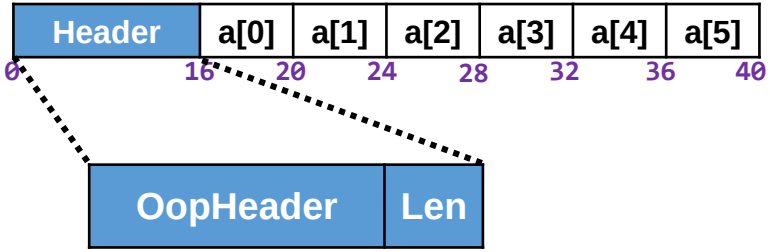
After optimization

```
cmp w3, w2
csel        w0, w3, w2, gt
```

Address Reshaping

```java
public static int loadInt(int[] arr, int n) {
    return arr[n];
}
```

-- Java array in memory --



Code Gen

array index

```
and x0, x2, #0xffffffff
lsl x0, x0, #2
add x0, x0, #0x10
ldr w0, [x1,x0]
```

array base

data offset

After optimization

array base

array index

```
add x0, x1, w2, uxtw #2
ldr w0, [x0,#16]
```

derived reference: base + index * 4

Intrinsic - bitCount

```java
/**
 * Returns the number of one-bits in the two's complement binary
 * representation of the specified {@code int} value.
 */
public static int bitCount(int m) {
    return Integer.bitCount(m);
}
```

```java
@MethodSubstitution
public static int bitCount(int value) {
    // Based on Warren, Hacker's Delight, slightly adapted
    // instruction.
    // Assuming the peephole optimizer optimizes all x - y
    // this takes 10 instructions.
    int x = value;
    x = x - ((x & 0xaaaaaaaa) >>> 1);
    x = (x & 0x33333333) + ((x & 0xcccccccc) >>> 2);
    x = (x + (x >>> 4)) & 0x0f0f0f0f;
    x = x + (x >>> 8);
    x = x + (x >>> 16);
    return x & 0x3f;
}
```

Graal implementation

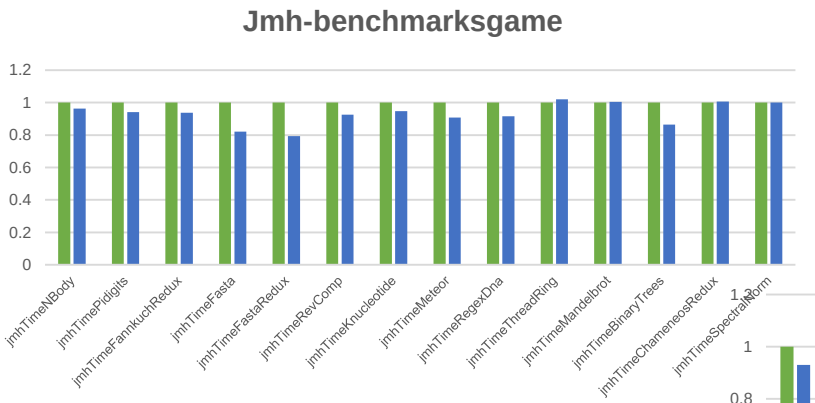
```arm
mov   x0, #0xaaaa
movk  x0, #0xaaaa, lsl #16
and   w0, w1, w0
lsr   w0, w0, #1
sub   w0, w1, w0
mov   x1, #0x3333
movk  x1, #0x3333, lsl #16
and   w1, w0, w1
mov   x2, #0xcccc
movk  x2, #0xcccc, lsl #16
and   w0, w0, w2
lsr   w0, w0, #2
add   w0, w1, w0
lsr   w1, w0, #4
add   w0, w1, w0
mov   x1, #0xf0f
movk  x1, #0xf0f, lsl #16
and   w0, w0, w1
lsr   w1, w0, #8
add   w0, w1, w0
lsr   w1, w0, #16
add   w0, w1, w0
and   w0, w0, #0x3f
```

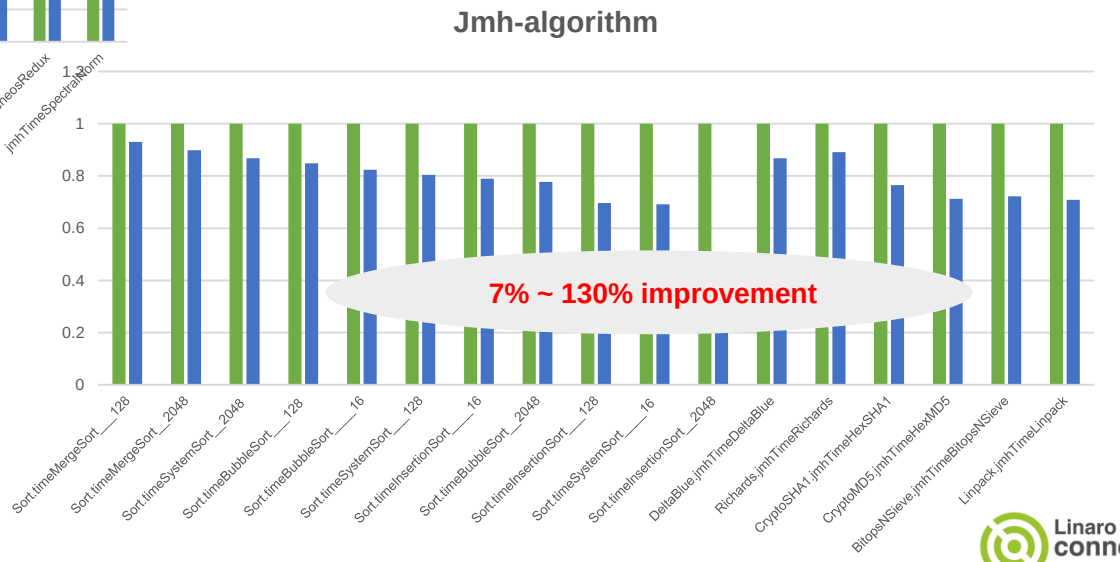
After optimization

```arm
fmov  s0, w1
cnt   v0.8b, v0.8b
addv  b0, v0.8b
mov   x0, v0.d[0]
```

Performance – with our optimizations (1)

Lower is better

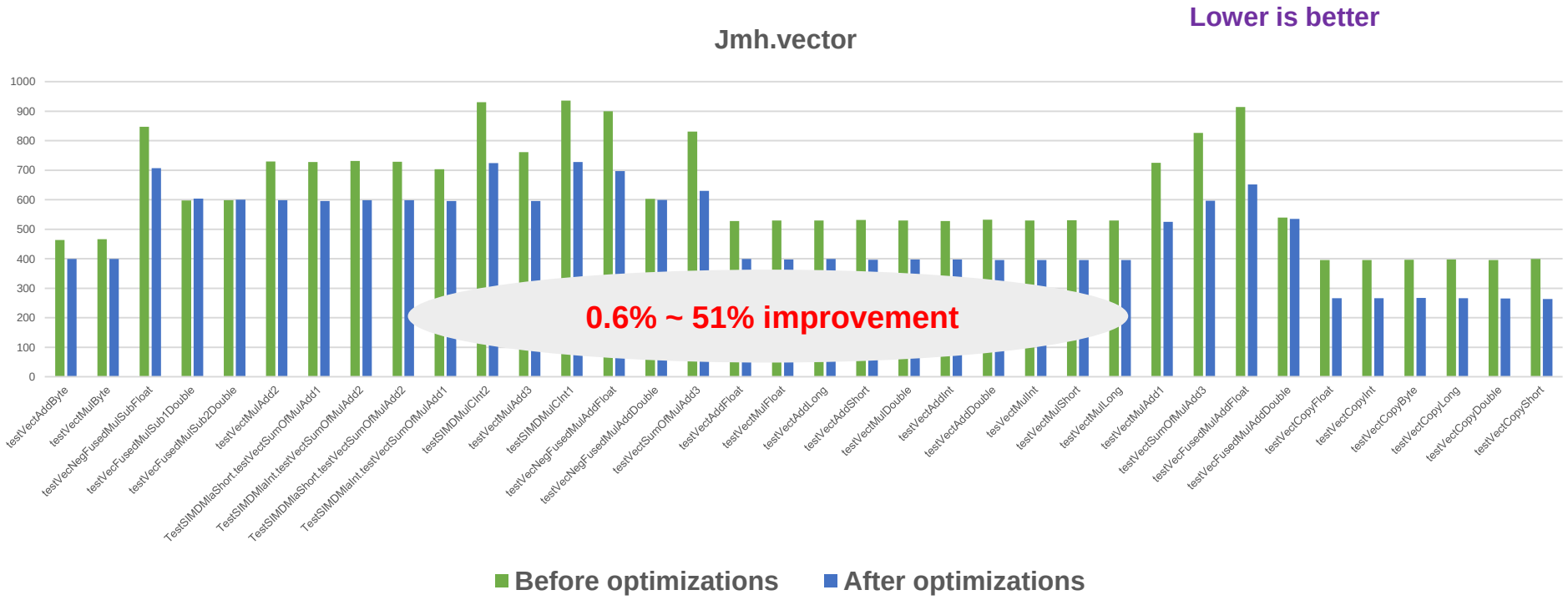




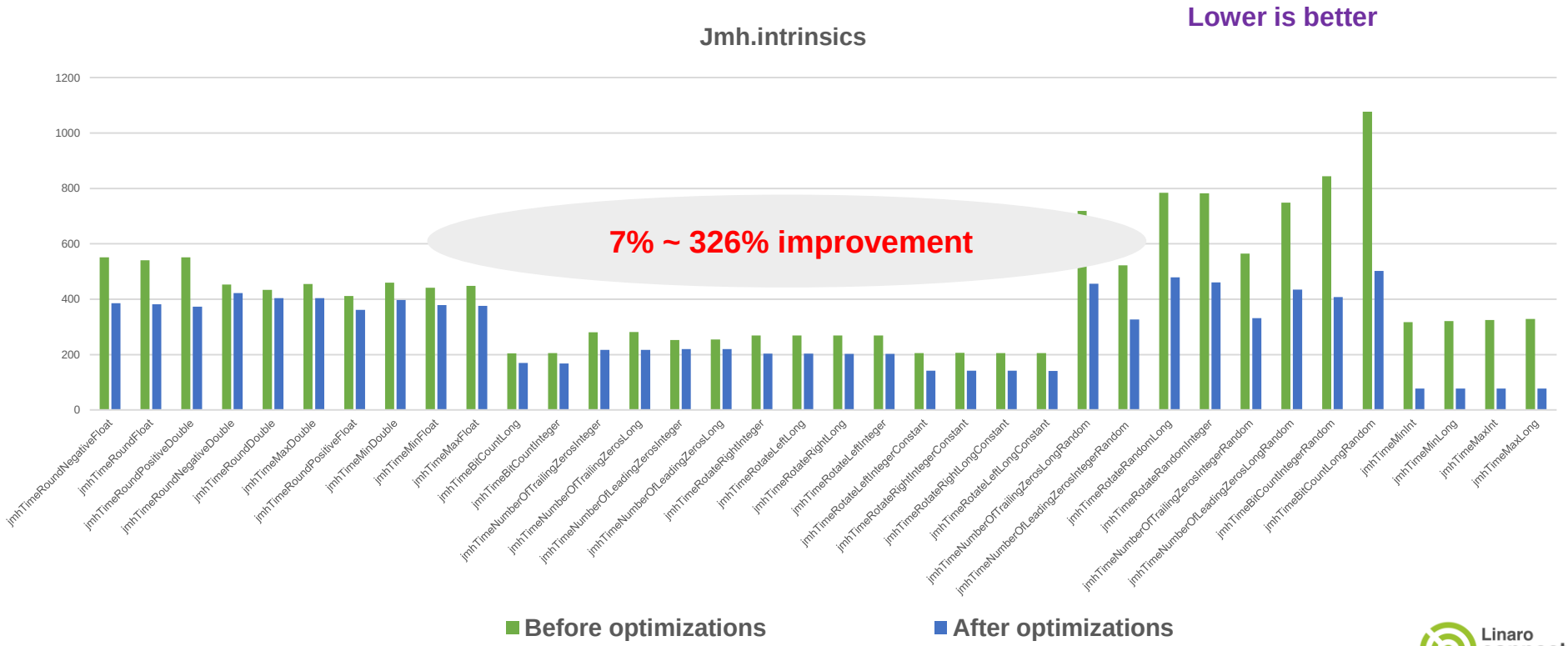
Before optimizations

After optimizations

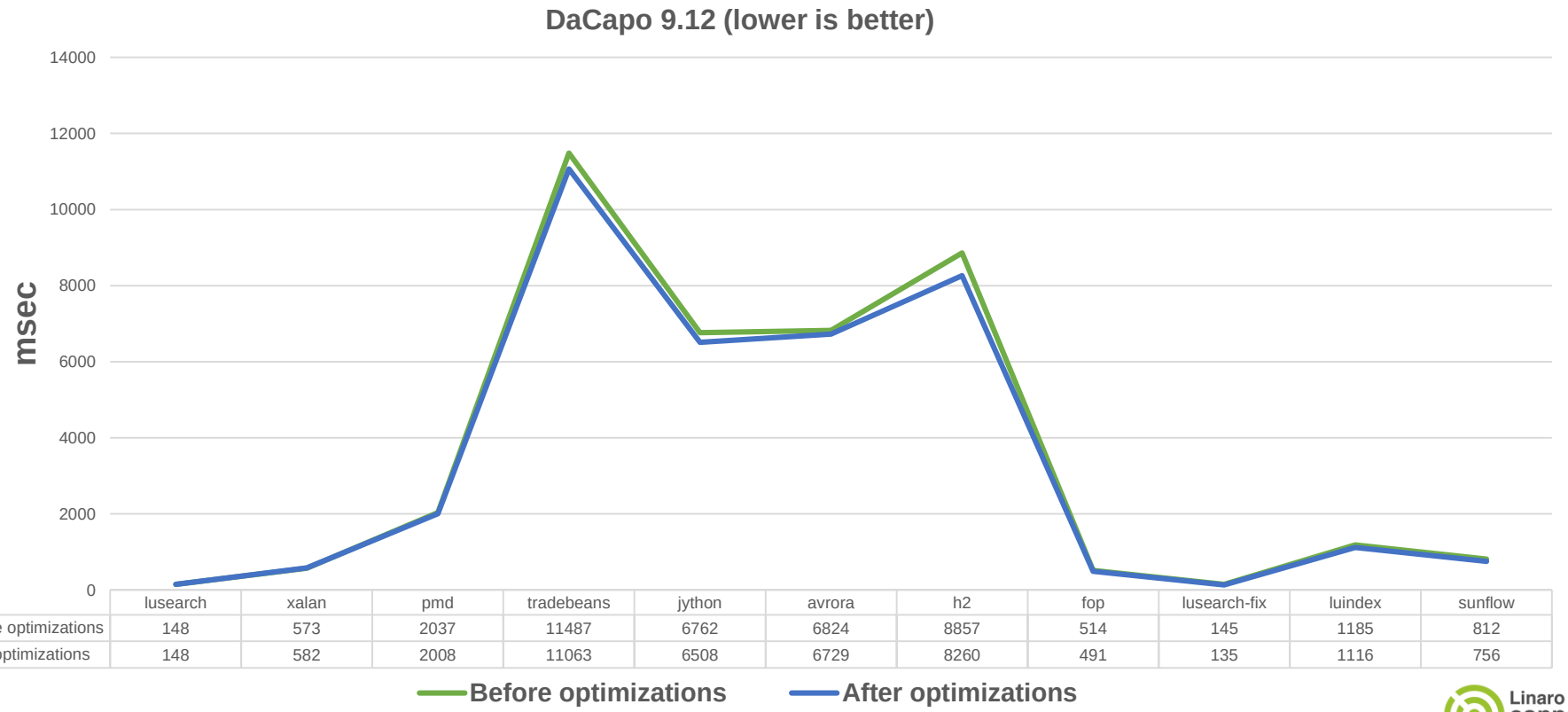
Performance – with our optimizations (2)



Performance – with our optimizations (3)



Performance – with our optimizations (4)



	lusearch	xalan	pmd	tradebeans	jython	avrora	h2	fop	lusearch-fix	luindex	sunflow
Before optimizations	148	573	2037	11487	6762	6824	8857	514	145	1185	812
After optimizations	148	582	2008	11063	6508	6729	8260	491	135	1116	756

The future work

Future work

- Loop related and other small optimizations
- ARM architecture new features(SVE, etc.)
- Instruction Selection (Missing match rules for AArch64 https://github.com/oracle/graal/issues/323
- Intrinsics (String, Unsafe, Math, etc.)
- Register Allocation & safepoint related optimizations

Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org

Why use Graal?

Compared to C2 and LLVM compiler, Graal is:

- Implemented in java.
- Modular, VM independent.
- Support other languages, not only for java.
- Support both dynamic and static compilation.

Graal's key optimizations that make its performance similar with C2:

- PEA
- Inlining

Partial Escape Analysis (PEA)

- EA is a method for determining the dynamic scope of an object, to decide whether to allocate it on the Java heap.
- PEA is a control flow sensitive EA, aimed at finding more escape analysis opportunities.

```java
public static class CacheKey {
    private final int idx;
    private final Object ref;

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

    public synchronized boolean equals(CacheKey other) {
        return idx == other.idx && ref == other.ref;
    }
}

public static CacheKey cacheKey = null;

public static Object testCacheSnippet(int idx, Object ref) {
    CacheKey key = new CacheKey(idx, ref);
    if (!key.equals(cacheKey)) {
        cacheKey = key;
        value = createValue(key);
    }
    return value;
}
```

allocate on heap

escape

A. An example C2 can not handle

PEA

```java
public static class CacheKey {
    private final int idx;
    private final Object ref;

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

    public synchronized boolean equals(CacheKey other) {
        return idx == other.idx && ref == other.ref;
    }
}

public static CacheKey cacheKey = null;

public static Object testCacheSnippet(int idx, Object ref) {
    if (!(idx == cacheKey.idx && ref == cacheKey.ref)) {
        CacheKey key = new CacheKey(idx, ref);
        cacheKey = key;
        value = createValue(key);
    }
    return value;
}
```

scalar replacement

allocate on heap

B. After analysis in Graal

What we have done – address reshaping

```
public static int loadInt(int[] arr, int n) {
    return arr[n];
}
```

Code Gen

array index

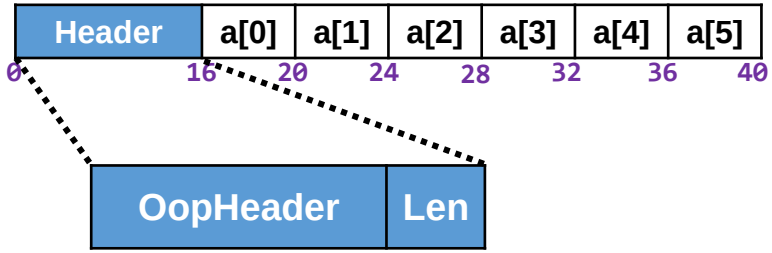
```
and x0, x2, #0xffffffff
lsl x0, x0, #2
add x0, x0, #0x10
ldr w0, [x1,x0]
```

array base

data offset

After optimization

-- Java array in memory --



array base

array index

safepoint is allowed

```
add x0, x1, w2, uxtw #2
ldr w0, [x0,#16]
```

derived reference: base + index * 4

OopMap(stackmap, metadata) describes:

- Oop – a GC root
- Derived_oop – base oop is described
- …