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:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacement for C2 compiler in Hotspot</td>
<td>JIT (Just-In-Time)</td>
</tr>
<tr>
<td>Hotspot AOT</td>
<td>AOT (Ahead-Of-Time)</td>
</tr>
<tr>
<td>Substrate VM</td>
<td>AOT</td>
</tr>
</tbody>
</table>

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

FrontEnd:
- High Tier
  - Inlining
  - Partial Escape Analysis
  - Lowering using Snippets
  - ...
- Mid Tier
  - Lock Elimination
  - Lowering using Snippets
  - ...
- Low Tier
  - Use Trap for Null Checking
  - Schedule
  - ...

BackEnd
- Register Allocation (Linear Scan)
- Peephole Optimizations
- Machine Code Emit

optimizations
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

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

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

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline, GVN, Constant propagation, Profile-guided Optimization (PGO), Canonicalization, Dead Code Elimination… (about 50 more)</td>
<td>Similar</td>
<td>Similar</td>
</tr>
<tr>
<td>Escape Analysis</td>
<td>Partial Escape Analysis</td>
<td>Non-partial EA</td>
</tr>
<tr>
<td>Lowering</td>
<td>Snippet</td>
<td>Non-snippet</td>
</tr>
<tr>
<td>Auto-vectorization</td>
<td>None</td>
<td>Superword-Level Parallelism (SLP) auto-vectorization</td>
</tr>
<tr>
<td>De-virtualization</td>
<td>Cache more virtual function calls</td>
<td>Cache fewer virtual function calls</td>
</tr>
</tbody>
</table>

### Back-End Optimizations on AArch64

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Selection</td>
<td>Naïve instruction selection, no match rules on AArch64 until we added some</td>
<td>BURS (Bottom Up Rewrite System), match rules on AArch64</td>
</tr>
<tr>
<td>Intrinsics</td>
<td>Missing a lot</td>
<td>Most implemented</td>
</tr>
<tr>
<td>Instruction scheduling</td>
<td>Less tuned scheduler</td>
<td>Architecture specific scheduler</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Linear Scan</td>
<td>Graph coloring</td>
</tr>
</tbody>
</table>
# Graal vs C2

## Mid-End Optimizations

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline, GVN, Constant propagation, Profile-guided Optimization (PGO), Canonicalization, Dead Code Elimination… (about 50 more)</td>
<td>Similar</td>
<td>Similar</td>
</tr>
<tr>
<td>Escape Analysis</td>
<td>Partial Escape Analysis</td>
<td>Non-partial EA</td>
</tr>
<tr>
<td>Lowering</td>
<td>Snippet</td>
<td>Non-snippet</td>
</tr>
<tr>
<td>Auto-vectorization</td>
<td>None</td>
<td>Superword-Level Parallelism (SLP) auto-vectorization</td>
</tr>
<tr>
<td>De-virtualization</td>
<td>Cache more virtual function calls</td>
<td>Cache fewer virtual function calls</td>
</tr>
</tbody>
</table>

## Back-End Optimizations on AArch64

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Selection</td>
<td>Naïve instruction selection, no match rules on AArch64 until we added some</td>
<td>BURS (Bottom Up Rewrite System), match rules on AArch64</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Missing a lot</td>
<td>Most implemented</td>
</tr>
<tr>
<td>Instruction scheduling</td>
<td>Less tuned scheduler</td>
<td>Architecture specific scheduler</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Linear Scan</td>
<td>Graph coloring</td>
</tr>
</tbody>
</table>
# Graal vs C2

## Mid-End Optimizations

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline, GVN, Constant propagation, Profile-guided Optimization(PGO), Canonicalization, Dead Code Elimination… (about 50 more)</td>
<td>Similar</td>
<td>Similar</td>
</tr>
<tr>
<td>Escape Analysis</td>
<td>Partial Escape Analysis</td>
<td>Non-partial EA</td>
</tr>
<tr>
<td>Lowering</td>
<td>Snippet</td>
<td>Non-snippet</td>
</tr>
<tr>
<td>Auto-vectorization</td>
<td>None</td>
<td>Superword-Level Parallelism (SLP) auto-vectorization</td>
</tr>
<tr>
<td>De-virtualization</td>
<td>Cache more virtual function calls</td>
<td>Cache fewer virtual function calls</td>
</tr>
</tbody>
</table>

## Back-End Optimizations on AArch64

<table>
<thead>
<tr>
<th></th>
<th>Graal</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Selection</td>
<td>Naive instruction selection, no match rules on AArch64 until we added some</td>
<td>BURS(Bottom Up Rewrite System), match rules on AArch64</td>
</tr>
<tr>
<td>Intrinsics</td>
<td>Missing a lot</td>
<td>Most implemented</td>
</tr>
<tr>
<td>Instruction scheduling</td>
<td>Less tuned scheduler</td>
<td>Architecture specific scheduler</td>
</tr>
<tr>
<td>Register Allocation</td>
<td>Linear Scan</td>
<td>Graph coloring</td>
</tr>
</tbody>
</table>
Performance on AArch64

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

![SPECjbb2015 (Higher is better)](image)

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.

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

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

---

---

After optimization

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

---

derived reference: base + index * 4

---

-- Java array in memory --
Intrinsic - bitCount

```java
/**
 * Returns the number of one-bits in the two's complement binary representation of the specified int value.
 */

public static int bitCount(int m) {
    return Integer.bitCount(m);
}
```

@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 & 0xc0cccccc) >> 2);
    x = (x + (x >>> 4)) & 0x0f0f0f0f;
    x = x + (x >>> 8);
    x = x + (x >>> 16);
    return x & 0x3f;
}

Graal implementation

After optimization

```assembly
mov x0, #0xaaaaa
movk x3, #0xaaaaa, 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, w2, w0
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
```
Performance – with our optimizations (1)

Before optimizations  
After optimizations

Lower is better

7% ~ 130% improvement
Performance – with our optimizations (2)

Jmh.vector

Lower is better

0.6% ~ 51% improvement

Before optimizations  After optimizations
Performance – with our optimizations (3)

Jmh.intrinsics

Lower is better

7% ~ 326% improvement

Before optimizations  After optimizations
Performance – with our optimizations (4)

DaCapo 9.12 (lower is better)

<table>
<thead>
<tr>
<th></th>
<th>lusearch</th>
<th>xalan</th>
<th>pmd</th>
<th>tradebeans</th>
<th>jython</th>
<th>avrora</th>
<th>h2</th>
<th>fop</th>
<th>lusearch-fix</th>
<th>luindex</th>
<th>sunflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before optimizations</td>
<td>148</td>
<td>573</td>
<td>2037</td>
<td>11487</td>
<td>6762</td>
<td>6824</td>
<td>8857</td>
<td>514</td>
<td>145</td>
<td>1185</td>
<td>812</td>
</tr>
<tr>
<td>After optimizations</td>
<td>148</td>
<td>582</td>
<td>2008</td>
<td>11063</td>
<td>6508</td>
<td>6729</td>
<td>8260</td>
<td>491</td>
<td>135</td>
<td>1116</td>
<td>756</td>
</tr>
</tbody>
</table>
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](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.

A. An example C2 can not handle

```
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;
    }
}
```

B. After analysis in Graal

```
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 (!cacheKey.equals(cacheKey)) {
            CacheKey key = new CacheKey(idx, ref);
            cacheKey = key;
            value = createValue(key);
        }
        return value;
    }
```
What we have done – address reshaping

After optimization

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

-- Java array in memory --

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

OopHeader | Len

array index

array base

data offset

OopMap(stackmap, metadata) describes:
- Oop – a GC root
- Derived_oop – base oop is described

...