

Jfokus 2020

Charlie Gracie
Java Engineering Group
at Microsoft

Current state of EA and its uses in the JVM

Overview

- Escape Analysis and Consuming Optimizations
- Current State of Escape Analysis in JVM JITs
- Escape Analysis and Allocation Elimination in Practice
- Stack allocation

Escape Analysis and Consuming Optimizations

What is Escape Analysis?

- **Escape Analysis** is a method for determining the dynamic scope of objects -- where in the program an object can be accessed.
- **Escape Analysis** determines all the places where an object can be stored and whether the lifetime of the object can be proven to be restricted only to the current method and/or thread.

Partial Escape Analysis

- A variant of Escape Analysis which tracks object lifetime along different control flow paths of a method.
- An object can be marked as not escaping along one path even though it escapes along a different path.

EA Consuming Optimizations

1. Monitor elision

- If an object does not escape the current method or thread, then operations can be performed on this object without synchronization

2. Stack allocation

- If an object does not escape the current method, it may be allocated in stack memory instead of heap memory

3. Scalar replacement

- Improvement to (2) by breaking an object up into its scalar parts which are just stored as locals

Current State of Escape Analysis in JVM JITs

HotSpot C2 EA and optimizations

- Flow-insensitive¹ implementation based on the following paper:

<https://b.gatech.edu/2GLKhk9>

- Optimizations that use EA results:
 - Monitor elision
 - Scalar replacement

Graal JIT EA and optimizations

- Partial Escape Analysis based on the following paper:

<https://bit.ly/37GMZTv>

- Optimizations that use EA results:
 - Monitor elision
 - Scalar replacement

OpenJ9 Testarossa EA and optimizations

- Partial Escape Analysis implemented here:

<https://bit.ly/2OhJLOV>

- Optimizations that use EA results:
 - Monitor elision
 - Discontiguous stack allocation (scalar replacement)
 - Stack allocation

Escape Analysis and Allocation Elimination in Practice

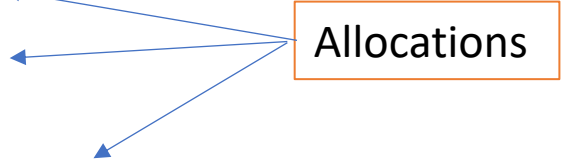
Example of non-escaping objects

```
public static int simple_math_new(int val1, int val2) {  
    Integer int1 = new Integer(val1);  
    Integer int2 = new Integer(val2);  
  
    return new Integer(int1.intValue()+int2.intValue());  
}
```

<https://github.com/charliegracie/jmh-benchmarks>

Example of non-escaping objects

```
public static int simple_math_new(int val1, int val2) {  
    Integer int1 = new Integer(val1);  
    Integer int2 = new Integer(val2);  
  
    return new Integer(int1.intValue()+int2.intValue());  
}
```



The diagram illustrates that the three objects created with the 'new' keyword (Integer int1, Integer int2, and the return Integer) are all allocations. Three blue arrows originate from a box labeled 'Allocations' and point to each of the 'new Integer' expressions in the code.

Example of non-escaping objects

JMH results for `simple_math_new` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	62,952,489	0
HotSpot with Graal	57,440,585	0
OpenJ9 Testarossa	62,473,794	0

Example of non-escaping objects

JMH results for `simple_math_new` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	62,952,489	0
HotSpot with Graal	57,440,585	0
OpenJ9 Testarossa	62,473,794	0



Limitations of scalar replacement

- Scalar replacement can fail because of a handful of reasons, but most notably it fails with the introduction of control flow

```
public int foo(MyClass[] array, int arg) {  
    MyClass obj;  
  
    if (arg > 127) {  
        obj = new MyClass(arg); // definition v1  
    } else {  
        obj = array[arg]; // definition v2  
    }  
  
    return obj.x; // v3 = Phi(v1, v2)  
}
```

A merge of the two definitions introduced on each side of the conditional

How common is this issue?

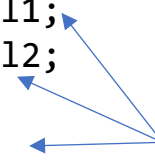
Can scalar replace

```
public static int simple_math_new(int val1, int val2) {  
    Integer int1 = new Integer(val1);  
    Integer int2 = new Integer(val2);  
  
    return new Integer(int1.intValue()+int2.intValue());  
}
```

Cannot scalar replace

```
public static int simple_math(int val1, int val2) {  
    Integer int1 = val1;  
    Integer int2 = val2;  
  
    return int1+int2;  
}
```

Auto boxing uses
`Integer.valueOf(int)`

The diagram consists of three blue arrows originating from a text box on the right and pointing to the right-hand side of the assignment statements in the 'Cannot scalar replace' code block. One arrow points to 'val1' in 'Integer int1 = val1;', another points to 'val2' in 'Integer int2 = val2;', and a third points to 'int1' in 'return int1+int2;'. This illustrates that the compiler uses autoboxing to convert the primitive 'int' values into 'Integer' objects.

Integer.valueOf(int) has the exact pattern, in that it pre-caches the *Integer* objects from -128 until 127

How common is this issue?

JMH results for `simple_math` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	51,107,977	2045
HotSpot with Graal	56,839,933	0 ¹
OpenJ9 Testarossa	51,020,520	0 ²

1. Graal has a special IR node for boxed Integers
2. OpenJ9 is stack allocating the objects

Simple change to the experiment

```
public static int simple_math_myobject_valueof(int val1, int val2) {  
    MyObject int1 = MyObject.valueOf(val1);  
    MyObject int2 = MyObject.valueOf(val2);  
  
    return int1.add(int2);  
}  
  
public MyObject valueOf(int val) {  
    if (0 <= val && val <= 2) {  
        return _cache[val];  
    } else {  
        return new MyObject(val);  
    }  
}
```

Simple change to the experiment

```
public static int simple_math_myobject_valueof(int val1, int val2) {  
    MyObject int1 = MyObject.valueOf(val1);  
    MyObject int2 = MyObject.valueOf(val2);  
  
    return int1.add(int2);  
}
```

```
public MyObject valueOf(int val) {  
    if (0 <= val && val <= 2) {  
        return _cache[val];  
    } else {  
        return new MyObject(val);  
    }  
}
```

← Simulate Integer.valueOf(int)

Simple change to the experiment

JMH results for `simple_math_myobject_valueof` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	52,764,203	2418
HotSpot with Graal	47,091,006	2160
OpenJ9 Testarossa	51,891,072	0 ¹

1. OpenJ9 is stack allocating the objects

Could we make scalar replacement work?

- We could pull the `obj.x` load up. C2 doesn't do this¹ and it gets complicated, e.g. side-effects.

```
MyClass obj;  
  
if ( arg > 127) {  
    obj = new MyObj(arg); // v1  
    return obj.x;  
} else {  
    obj = array[abc]; // v2  
    return obj.x;  
}
```

Stack Allocation

Considering the previous example...

- The value is a proper object on each side of the control flow, no issues with merging

```
MyClass obj;  
  
if (arg > 127) {  
    obj = [stack alloc] MyClass(arg); // v1  
} else {  
    obj = array[arg]; // v2  
}  
  
return obj.x; // v3 = Phi(v1, v2) ←
```

obj is a merge of the two definitions introduced on each side of the conditional

Stack allocate if scalar replacement fails

- We can do stack allocation of the whole object
 - The object shape is preserved, it just lives on the stack instead of the heap
- Does it even make sense to stack allocate?

```
public int addSome(int val1, int val2) {  
    Integer int1 = val1;  
    Integer int2 = val2;  
  
    for (int i = 0; i < val2; i++) {  
        int1 += int2;  
    }  
  
    return int1;  
}
```


Boxing call using Integer.valueOf()



Stack allocate if scalar replacement fails

- We can do stack allocation of the whole object
 - The object shape is preserved, it just lives on the stack instead of the heap
- Does it even make sense to stack allocate?

```
public int addSome(int val1, int val2) {  
    Integer int1 = val1;  
    Integer int2 = val2;  
  
    for (int i = 0; i < val2; i++) {  
        int1 += int2;  
    }  
  
    return int1;  
}
```




Allocation on every loop iteration

Stack allocate if scalar replacement fails

- We can do stack allocation of the whole object
 - The object shape is preserved, it just lives on the stack instead of the heap
- Does it even make sense to stack allocate?

```
public int addSome(int val1, int val2) {  
    Integer int1 = val1;  
    Integer int2 = val2;  
  
    for (int i = 0; i < val2; i++) {  
        int1 += int2;  
    }  
  
    return int1;  
}
```



Unboxing on return keeps int1
within the method context

What if C2 had stack allocation?

```
public static int simple_math_myobject_valueof(int val1, int val2) {  
    MyObject int1 = MyObject.valueOf(val1);  
    MyObject int2 = MyObject.valueOf(val2);  
  
    return int1.add(int2);  
}
```

What if C2 had stack allocation?

JMH results for `simple_math_myobject_valueof` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	52,764,203	2418
HotSpot with Graal	47,091,006	2160
OpenJ9 Testarossa	51,891,072	0

What if C2 had stack allocation?

JMH results for `simple_math_myobject_valueof` using JDK11

JIT Compiler	Op/s	GC Count
HotSpot C2	52,764,203	2418
HotSpot with Graal	47,091,006	2160
OpenJ9 Testarossa	51,891,072	0
HotSpot C2 + Stack Allocation	56,980,426	0

How did we implement this in C2

- We enhanced the Escape Analysis in C2 to detect the cases we can safely stack allocate
 - Not every non-escaping object can be allocated on the stack
- We implemented stack allocation in Macro Expansion, where we remove everything but the stack allocated path

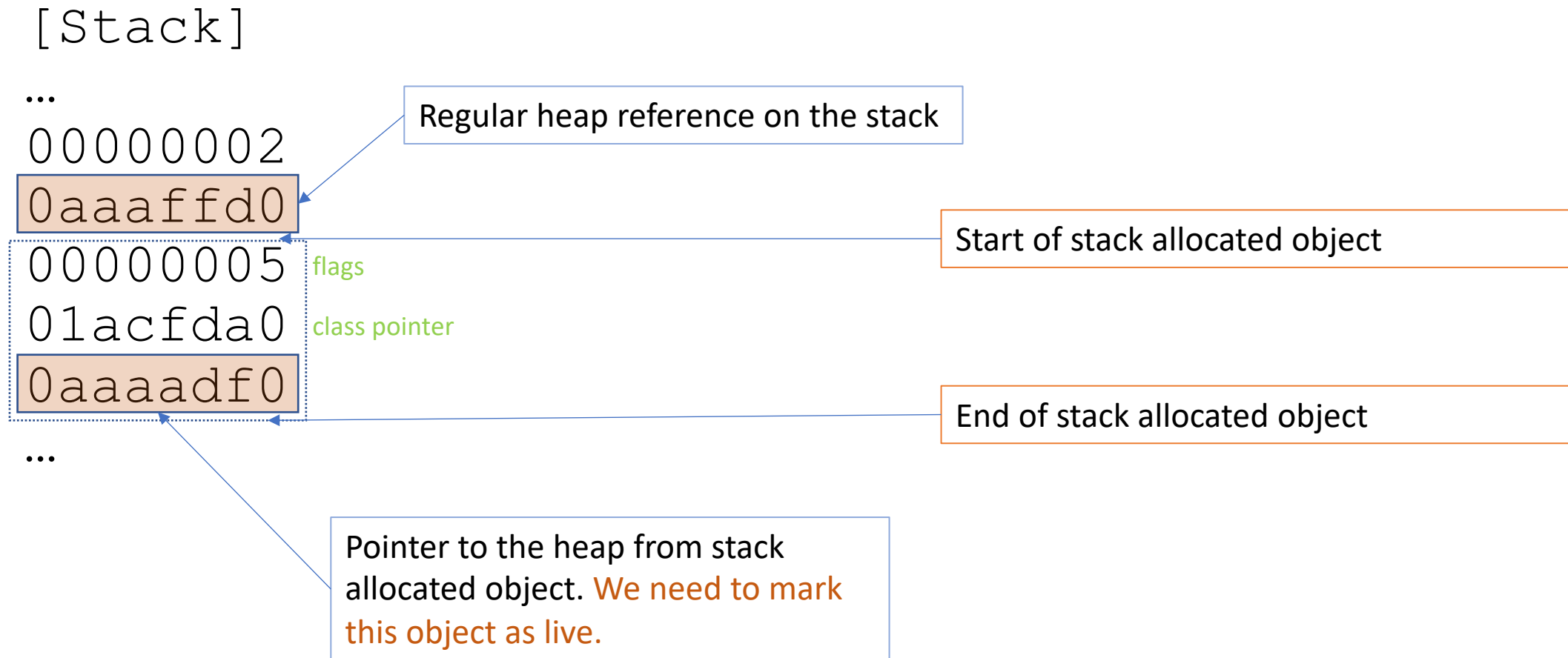
BoxLockNode to the rescue

- We needed a way to reserve stack memory for the allocation
- We leveraged BoxLockNode (meant for monitor slots) to communicate the stack allocation position from the optimizer to the codegen
 - Stack allocated objects are located at the bottom of the frame, right after the locals/spills

We had to...

- Extend the GC root scanning to understand stack allocated objects
- Detect overlapping live ranges of objects in loops
- Remove write barriers on stack allocated objects
- Implement support for “heapification” of objects at deoptimization time
 - Can happen at any safepoint that the allocation can reach

GC root scanning of stack allocated objects



Overlapping live object ranges

```
[SA] Integer value = 0;
```

Let's call this definition v1

```
while (some condition) {
```

```
  [SA] Integer result = call();
```

v3 = Phi(v1, v2)

```
  if (!value.equals(result)) {
```

```
    value = result;
```

Let's call this definition v2

```
  }
```

```
}
```

```
sum += value;
```

v4 = Phi(v1, v2, v3)

Definition v2 makes [address of value] == [address of result]

Current Limitations

- We don't stack allocate objects in methods with monitors
- We don't allow stack allocated objects to be stored into other objects
 - Heap parent would mean an escaping object
 - Stack allocated to stack allocated would require more analysis

Current Limitations

- No compressed oop support yet
 - Stack allocated object pointers cannot be compressed / decompressed¹
- We don't stack allocate arrays
- Project Loom fast relocation will need special considerations

Performance improvements

- We have measured performance on Scala DaCapo benchmarks and GraalVM Renaissance benchmarks
 - Stack allocation is always a win, no regressions

Benchmark	Percent Improvement
DaCapo tmt	15%
Renaissance neo4j-analytics	45%
Renaissance db-shootout	10%
Renaissance fj-kmeans (spark)	5%
Renaissance future-genetic	10%
Renaissance movie-lens (spark)	5%

When and where can we see this patch?

- We made the prototype on top of JDK11 and we are in process of cleaning up the code and migrating it to tip
- As soon as we are done with the migration, we'll post the patch on the hotspot-compiler-dev mailing list

Next steps

- Stabilize the prototype and migrate the code to tip
 - We still have few bugs with GC and deoptimization
- Remove the limitations one by one
- Extend GC support to cover more modes
- Look for opportunities in more workloads

Liked the presentation? Stay connected on the *hotspot-compiler-dev* mailing list or even better, help us with reviewing the patch :)

Charlie.Gracie@microsoft.com @crgracie

Nikola.Grcevski@microsoft.com @vivaboredom

@JavaAtMicrosoft