New tricks of the GraalVM

Tõnis Pool
University of Tartu
tonis.pool@gmail.com
Supervised by Oleg Batrashev

Abstract

This report gives a high level overview of the Graal JIT compiler developed in Oracle Labs in collaboration with the Institute for System Software at the Johannes Kepler University. Graal is written in Java and aims at replacing or complementing the HotSpot server and client compilers. I'll introduce the main concepts of how Graal works and bring out the biggest differences to the current HotSpot compilers.

1. Introduction

Of all the components in a managed runtime none have a larger impact on the perfomance than the just-in-time (JIT) compiler[1]. Managed runtimes usually start off by implementing an interpreter of sorts for the target language or some intermediate language like bytecode for Java. But interpreters could never compete with the performance of machine code that systems programming languages like C for example produce when compiled.

JIT compilation was the technique that really made it feasible to have fast managed runtimes for languages with higher abstractions than systems programming languages. Depending on your application and use case Java nowadays can be as fast or faster than C[2]. Though JIT compilers are necessary they are also difficult to implement and modify afterwards.

Graal is a new extensible high-performance optimizing JIT compiler for Java. It can be integrated with different virtual machines and has been integrated with the Maxine VM and the HotSpot VM. It's written in Java and has a modular design - the compiler consits of around 100 different modules. The term GraalVM is used to denote a HotSpot virtual machine configured with Graal[3].

2. Architecture

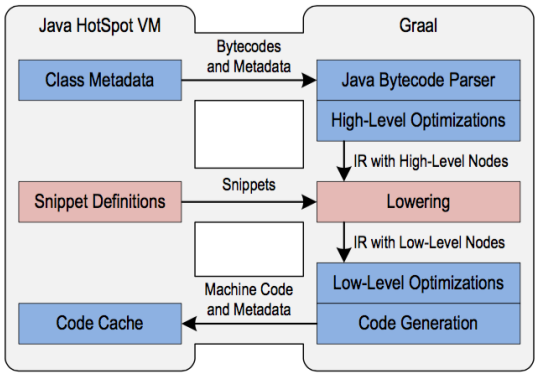

Figure 1: *GraalVM architecture overview*

Figure 1 shows the high level architecture overview of how the compiler interfaces with the VM[4]. Naturally the compiler needs to query the VM for metadata/information, thus the VM is required to implement a certain set of interfaces that provide the needed values. Concrete examples are (a non-exhaustive list):

1. *MetaAccessProvider* - Provides access to the metadata of a class typically present in a Java class file.
2. *CodeCacheProvider* - Access to code cache related details and requirements.
3. *ConstantReflectionProvider* - Reflection operations on values represented as constants.
4. *StackIntrospection* - Accesses the current stack.

Graal can be configured as a replacement for the HotSpot server compiler or as an additional compiler that compiles only upon special requests. The last setup is used by the Truffle language implementation framework, that leverages graal to achieve great performance for new programming languages[5].

2.1. Compilation Pipeline

Compilation follows the following logical path within the Graal Compiler[4]:

1. Bytecode parsing - The compilation target method's bytecode is parsed into a Graal's high-level intermediate representation (IR), which is discussed more in Section 3.
2. Front end - Works on the graph based IR and is further divided into the following
   (a) High Tier - Performs optimisations such as method inlining and partial escape analysis which are discussed further in Sections 4 and 5
   (b) Mid Tier - Performs some memory optimisations and lock coarsening for example.
   (c) Low Tier - Final cleanups and a smaller optimisations such as using trapping signals for null cheks.
3. Back end - Deals with register allocation and machine code generation.

3. Graal IR

Graal IR consists of two interposed directed graphs - the control flow graph and the data flow graph. Data flow graph consists of nodes pointing "upwards" to the nodes that produce its operands. Control flow graph consists of nodes pointing "downwards" to their different possible successors[6].

The Graal IR is in Static Single Assignment (SSA) form. SSA means that each variable is assigned exactly once, and every variable is defined before it is used. All variables that are assigned multiple times are split into versions. If the multiple assignments happen in different branches then special ϕ nodes are inserted at branching join nodes, that take the different values coming from the 2 or more control flows and choose the value of the variable from whose edge the ϕ node was accessed from. The resulting value is put into yet another versioned variable[7]. This is illustrated in how code in Listing 1 is turned into SSA form in Listing 2.

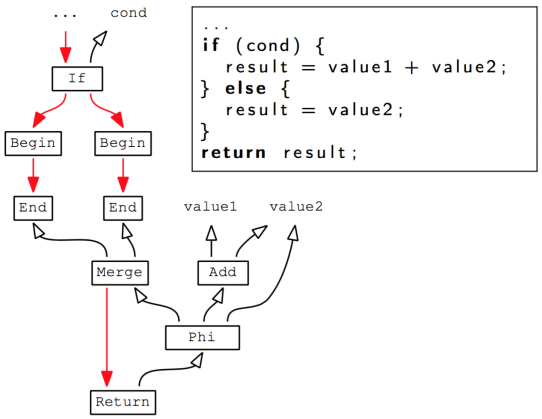

Figure 2: *Graal IR with control-flow edges pointing down and data-flow edges pointing up*

Not all nodes are fixed into the IR right from the start. Control flow splits and merges form a backbone around which most other nodes float. Floating nodes are only constrained by their data flow inputs and additionally memory dependencies. Floating nodes allow a great deal of freedom of movement for different optimisations such as transferring code that doesn't depend on loop variables outside of loops so that they wouldn't be executed needlessly[8].

4. Inlining

Method inlining is the practice of replacing a method call within a method with the code from the inlined method. Instead of calling another method its code is directly copied over to the caller. As JIT compilation is the biggest performance booster for managed language runtimes, method inlining is the "mother of all

Listing 1: *Example code before SSA*

```
1 if (flag) {
    x = 0
3 } else {
    x = 42
5 }
  println x
```

Listing 2: *Example code in SSA*

```
  if (flag) {
2   x1 = 0
  } else {
4   x2 = 42
  }
6 x3 = phi(x1, x2)
  println x3
```

optimisations" as it enables additional optimisations on the larger chunk of code that was previously "hidden" behind method barriers[9].

HotSpot *client* and *server* compilers both inline at the stage of bytecode parsing, just as if the bytecode of the called method would have been directly in the caller. This approach has several drawbacks[10]:

1. Complex parsing - the compiler has to make the decision, whether to inline or not, very early. This forces other optimisations to be done during bytecode parsing as well, which complicates parsing.

2. No way to correct later - When bytecode is parsed, method inlining decisions are done and it's very hard to either revert the inlining decision or do additional inlining during later stages of the compilation. As an example other optimisations, such as escape analysis (discussed further in Section 5), might find additional inlining desirable.

3. No way to cache the results - Inlining other methods while parsing the bytecode can be seen as introducing side effects to the method, when another method would like to inline the same method it can't simply copy the parsed bytecode, as it may contain unwanted instructions from previously inlined method calls.

To address these concerns Graal uses a technique known as *late inlining*, which simply means that it's possible to delay the decisions to inline methods. This has the potential to make bytecode parsing a pure function, meaning that with same input it outputs the same IR that can be cached for reuse. Inlining a method then is nothing more than replacing the invoke node in the IR with the IR of the corresponding method.

5. Partial Escape Analysis

Escape analysis is the optimisation of instead allocating an object on the JVM shared heap it's allocated on the current thread's stack or its fields pushed to registers directly avoiding an allocation all together - the latter is a technique known as *Scalar Replacement*.

HotSpot supports escape analysis and it defines 3 different escape levels[11]:

1. *NoEscape* - Object is effectively method-local and allocation can be totally removed with fields replaced via scalar replacement.

2. *MethodEscape* - Object escapes the current method but not the thread, which means that they can be allocated on the stack with all synchronisation removed.

3. *GlobalEscape* - Object escapes potentially to other threads.

Escape analysis in HotSpot works by building so called *equi-escape sets* of objects where all elements in the set share the same - maximum of all objects - escape state. Initially all objects are in different sets. It then analyses all the operations in the method merging sets when an object in one set is assigned to a field of an

Listing 3: *Example code before partial escape analysis*

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

Listing 4: *Example code after partial escape analysis*

```
Object get(int idx, Object ref) {
  Key tmp = cacheKey;
  if (idx == tmp.idx
      && ref == tmp.ref) {
    return cacheValue;
  } else {
    Key key = new Key(idx, ref);
    cacheKey = key;
    cacheValue = createValue (...);
    return cacheValue;
  }
}
```

object in another set for example. When one of the objects in a set escapes, via assignment to a static variable for example, the whole set of objects escapes with it[12].

The algorithm outlined above makes a global decision about the escapability of objects in a method. The main idea behind *partial escape analysis* is to analyse the escapability of objects for individual branches. Meaning if an object only escapes in one branch of the program it can still be replaced with scalar replacement in all other, hopefully reducing the number of allocations in runtime[12].

This is illustrated in Listing 3 and 4, where the newly created `Key` object only escapes to the static field `cacheKey` when it's different from the current `cacheKey`. If mostly it is equal then it's beneficial to avoid the object allocation always and do it only if needed. In this example partial escape analysis also leverages late inlining to inline the `Key#equals` method, which allows it to do scalar replacement of the fields of `Key` class and simplify the method call to method arguments equals checks.

6. Snippets

Finally we arrive at the concept that really makes having a separate JIT compiler written in a high level language possible - snippets. Snippets can be seen as the glue that ties the compiler and the JVM together. The problem snippets solve is that of how to represent the low-level semantics of a high-level operation using a high-level programming language. As already mentioned in the introduction Graal is a compiler written in Java that compiles Java bytecode into native code. But bytecode contains high level operations such as `INSTANCEOF` or `NEW` instructions that need to be somehow mapped to low level machine instructions - this process is called *lowering*[13].

HotSpot tackles the lowering problem by inserting hand-crafted assembly templates in the *client* compiler and replacing high-level operations within the *server* compiler with hand-assembled IR graphs directly[13]. This has the downside of complicating porting to new architectures - as all the assembly templates need to be rewritten - and the hand-assembled IR graphs being difficult to change or analyse. In comparison Snippets are written in Java with a few limitations such as the object creation snippet cannot use the `new` keyword itself. This makes them more approachable to people who are not familiar with assembly code or C++.

A snippet is a static Java method with the Java annotation `@Snippet`. The method can take various inputs that are substituted in at different times during the snippet lifecycle, discussed shortly. The body of the method expresses the semantics of the instruction being lowered. It can call additional methods, have if and loop statements etc.

Listing 5: *Simplified DeoptimizeNode together with an example snippet*

```
public final class DeoptimizeNode extends AbstractDeoptimizeNode {
  protected final DeoptimizationAction action;
  protected final DeoptimizationReason reason;

  public DeoptimizeNode(DeoptimizationAction action,
      DeoptimizationReason reason) {
    this.action = action;
    this.reason = reason;
  }

  @NodeIntrinsic
  public static native void deopt(
        @ConstantNodeParameter DeoptimizationAction action,
        @ConstantNodeParameter DeoptimizationReason reason);
}

@Snippet
public static Object mySnippet(Object object,
    @ConstantParameter boolean nullSeen) {
  if (object == null && !nullSeen)
    DeoptimizeNode.deopt(InvalidateReprofile, NullCheckException);
  ...
}
```

As can be seen from Figure 1 snippets are provide by the VM so they are VM specific meaning they have access to VM internal data structures such as object headers and type information. To make them feasible in practice snippets are also compiler specific as they have to influence the generated IR.

6.1. Snippet Lifecycle

Snippets go through the following lifecycle[13]:

1. *Preparation* - As snippets are regular Java code they are compiled into bytecode like all other source code. Similarly to all bytecode snippets are parsed into the same Graal IR with some additional processing steps such as node intrinsification and exhaustive inlining. This step is performed at once per snippet as the resulting IR is cached after parsing.
2. *Specialization* - Snippets can take configuration parameters that will be replaced with constants during snippet specialisation phase so that parts of the snippet logic can be optimised away. Such parameters are either annotated with `@ConstantParameter` or `@VarargsParameter` and an example would be wether the profiler has seen a null value for this `INSTANCEOF` check during interpretation or what array of types has it seen pass through the instruction. Specialisation is done only once for the unique combination of values bound to the constant parameters as the results of each specialisation is also cached. Node intrinsification is also repeated in case some intrinsifications were deferred until certain values become constant.
3. *Instantiation* - To really lower a high level instruction its corresponding node in the

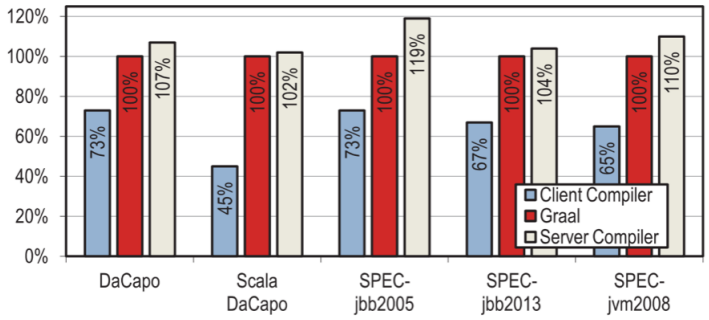

Figure 3: *Graal compiler performance compared to HotSpot compilers*

IR is simply replaced with a duplicate of the specialised snippet graph. It's executed once per snippet-lowerable IR node.

6.2. Node Intrinsics

Snippets expose an inherent problem in them as they need to be written in readable high level Java code while not compromising the resulting machine code. The problem is adressed by introducing *node intrinsics* - a simple way to insert compiler specific IR nodes by method calls. Node intrinsic is a `static native void` method that is annotated with `@NodeIntrinsic(SomeNode.class)` that has the node class as parameter to the annotation. Every call to such a method is replaced with the IR node specified by the `@NodeIntrinsic` annotation during node intrinsification. Node intrinsics can introduce additional so called medium-level nodes that are again lowered by additional snippets. To facilitate this Graal invokes the lowering phase for each compilation tier. Methods are marked native as that way they don't have to declare a body, which makes sense as they are really never invoked[13].

As an example snippets might need to deoptimize when information gathered during profiling turns out to be false, in such places snippet's have code to check the assumptions and call the `deopt(action, reason)` method. Listing 5 shows a simplified DeoptimizeNode in Graal that has a `@NodeIntrinsic` annotated method. It's used from a fictional snippet `mySnippet` that checks if an object is null, but shouldn't be, and calls the `deopt` method. This invoke node in the snippet IR is substituted with a DeoptimizeNode during the node intrinsification step.

7. Summary

Figure 3 shows the state of the Graal compiler compared to HotSpot *client* and *server* compilers as of June 2015[13]. It's clear that not only is a JIT compiler written in Java feasible, but it's also high performance. Graal JIT compiler is better than the *client* compiler in all tested benchmarks, and falls only shortly behind the *server* compiler in most.

Graal builds upon decades of compiler theory and advancement done in the HotSpot compilers adding its own refinements on top and porting everything to Java. It is a great piece of engineering and the result of years of research and experimentation. Personally I've never been able to understand much of how JIT compilers do their magic, but after reading the Graal source code, it becomes a bit more understandable and less magical.

References

[1] S. Oaks, *Java Performance: The Definitive Guide*. O'Reilly Media, 2014.

[2] C. Click, "Java vs. c performance....again.." `http://www.`

cliffc.org/blog/2009/09/06/java-vs-c-performanceagain/, September 2009.

[3] "http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html." http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html.

[4] C. Wimmer, "Graal tutorial." http://lafo.ssw.uni-linz.ac.at/papers/2015_CGO_Graal.pdf, February 2015.

[5] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, (New York, NY, USA), pp. 187–204, ACM, 2013.

[6] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck, "An intermediate representation for speculative optimizations in a dynamic compiler," in *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, (New York, NY, USA), pp. 1–10, ACM, 2013.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, (New York, NY, USA), pp. 1–11, ACM, 1988.

[8] C. Click, "Global code motion/global value numbering," *SIGPLAN Not.*, vol. 30, pp. 246–257, June 1995.

[9] D. Hawkins, "Jvm mechanics," 2 2015. As Presented at Silicon Valley Java User Group by Azul Systems [Accessed: 2015 11 21].

[10] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger, "Compilation queuing and graph caching for dynamic compilers," in *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, (New York, NY, USA), pp. 49–58, ACM, 2012.

[11] T. Kotzmann and H. Mössenböck, "Escape analysis in the context of dynamic compilation and deoptimization," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, (New York, NY, USA), pp. 111–120, ACM, 2005.

[12] L. Stadler, T. Würthinger, and H. Mössenböck, "Partial escape analysis and scalar replacement for java," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 165:165–165:174, ACM, 2014.

[13] D. Simon, C. Wimmer, B. Urban, G. Duboscq, L. Stadler, and T. Würthinger, "Snippets: Taking the high road to a low level," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 20:20:1–20:20:25, June 2015.