The Graal Compiler

Design and Strategy
work in progress

Thomas Würthinger [*], Lukas Stadler [§], Gilles Duboscq [*]

Created: April 27, 2011

Abstract The basic motivation of graal is to show the advantage, both in terms of development effort and runtime speed, a compiler written in Java can deliver to a C++-based virtual machine. This document contains information about the proposed structure and design of the Graal Compiler, which is part of the Maxine project.

1 Project Source Structure

In order to have clear interfaces between the different parts of the compiler, the code will be divided into the following source code projects:

Graph contains the abstract node implementation, the graph implementation and all the associated tools and auxiliary classes.

Nodes contains the node implementations, ranging from high-level to machine-level nodes.

GraphBuilder contains helpers for building graphs from java bytecodes and other source representations.

Assembler contains the assembler classes that are used to generate the compiled code of methods and stubs.

Optimizations contains all the optimizations, along with different optimization plans.

GraalCompiler contains the compiler, including:
- Handling of compilation phases.
- Compilation-related data structures.
- Implementation of the *compiler interface* (CI).
- Register allocation.
- Machine code creation, including debug info.
- Debug output and compilation observation.
- Compiler options management.

[*]Oracle, [§]Johannes Kepler University, Linz

2 Initial Steps

- Restructuring of the project to include the compiler and the modified HotSpot code within one repository. The CRI project will remain in the Maxine repository, because it will remain mostly unchanged.
- Stripping optimizations from the existing compiler, they will be reimplemented later on using the new infrastructure.
- Creating Node and Graph classes, along with the necessary auxiliary classes.
- Writing documentation on the design of the compiler.
- Use the Node class as the superclass of the existing Value class.
- Identify (and later: remove) extended bytecodes.
- Implement the new frame state concept.
- Remove LIR - in the long run there should only be one IR, which will be continually lowered until only nodes that can be translated into machine code remain.

3 Nodes and Graphs

The most important aspect of a compiler is the data structure that holds information about an executable piece of code, called *intermediate representation* (IR). The IR used in the Graal Compiler (simply refered to as *the compiler* in the rest of this document) was designed in such a way as to allow for extensive optimizations, easy traversal, compact storage and efficient processing.

3.1 The Node Data Structure

- Each node is always associated with a graph.

- Each node has an immutable id which is unique within its associated graph.
- Nodes represent either operations on values or control flow operations.
- Nodes can have a data dependency, which means that one node requires the result of some other node as its input. The fact that the result of the first node needs to be computed before the second node can be executed introduces a partial order to the set of nodes.
- Nodes can have a control flow dependency, which means that the execution of one node depends on some other node. This includes conditional execution, memory access serialization and other reasons, and again introduces a partial order to the set of nodes.
- Nodes can only have data and control dependencies to nodes which belong to the same graph.
- Control dependencies and data dependencies each represent a *directed acyclic graph* (DAG) on the same set of nodes. This means that data dependencies always point upwards, and control dependencies always point downwards. Situations that are normally incurr cycles (like loops) are represented by special nodes (like LoopEnd).
- Ordering between nodes is specified only to the extent which is required to correctly express the semantics of a given program. Some compilers (notably the HotSpot client compiler) always maintain a complete order for all nodes (called *scheduling*), which impedes advanced optimizations. For algorithms that require a fixed ordering of nodes, a temporary schedule can always be generated.
- Both data and control dependencies can be traversed in both directions, so that each node can be traversed in four directions:
  - *inputs* are all nodes that this node has data dependencies on.
  - *usages* are all nodes that have data dependencies on this node, this is regarded as the inverse of inputs.
  - *successors* are all nodes that have a control dependency on this node.
  - *predecessors* are all nodes that this node has control dependencies on, this is regarded as the inverse of successors.
- Only inputs and successors can be changed, and changes to them will update the usages and predecessors.
- The Node class needs to provide facilities for subclasses to perform actions upon cloning, dependency changes, etc.
- Nodes cannot be reassigned to another graph, they are cloned instead

3.2 The Graph Data Structure

- A graph deals out ids for new nodes and can be queried for the node corresponding to a given id.
- Graphs can manage side data structures, which will be automatically invalidated and lazily recomputed whenever the graph changes. Examples for side data structures are dominator trees and temporary schedules. These side data structures will usually be understood by more than one optimization.
- Graphs are

4 Frame States

Frame states capture the state of the program, in terms of the source representation (e.g., Java state: local variables, expressions, ...). Whenever a safepoint is reached or the a deoptimization is needed a valid frame state needs to be available. A frame state is valid as long as the program performs only actions that can safely be reexecuted (e.g., operations on local variables, etc.). Thus, frame states need only be generated for bytecodes that can not be reexecuted: putfield, astore, invoke, monitorenter/exit, ...

Within the node graph a frame state is represented as a node that is fixed between the node that caused it to be generated (data dependency) and the node that will invalidate it (control dependency).

Deopmization nodes are not fixed to a certain frame state node, they can move around freely and will always use the correct frame state. At some point during the compilation deoptimization nodes need to be fixed, which means that appropriate data and control dependencies will be inserted so that it can not move outside the scope of the associated frame state. This will generate deoptimization-free zones that can be targeted by the most aggressive optimizations.

Frame states should be represented using a delta-encoding. This will make them significantly smaller and will make inlining, etc. much easier. In later compilation phases unnecessary frame states might be removed, using a mark-and-merge algorithm.

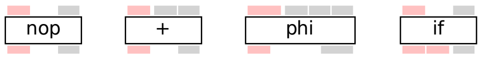
5 Graph Building

- The graph built by the initial parser (called *GraphBuilder*) should be as close to the source representation (bytecode, ...) as possible.

- All nodes should be able to immediately lower themselves to a machine-level representation. This allows for easier compiler development, and also leads to a compiler that is very flexible in the amount of optimizations it performs (e.g. recompilation of methods with more aggressive optimizations).
-

6 Graphical Representation

The graphs in this document use the following node layout:



Red arrows always represents control dependencies, while black arrows represent data dependencies: