The Graal Compiler

Design and Strategy

work in progress

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

Created: May 5, 2011

Abstract The Graal compiler (simply referred to as *the compiler* in the rest of this document) aims at improving C1X, the Java port of the HotSpot client compiler, both in terms of modularity and peak performance. The compiler should work with the Maxine VM and the HotSpot VM. This document contains information about the proposed design and strategy for developing the compiler.

1 Context

In 2009, the Maxine team started with creating C1X, a Java port of the HotSpot client compiler, and integrate it into the Maxine VM. Part of this effort, was the development of a clear and clean compiler-runtime interface that allows the separation of the compiler and the VM that enables the use of one compiler for multiple VMs. In June 2010, we started integrating C1X into the HotSpot VM and we called the resulting system Graal VM. Currently, the Graal VM is fully functional and runs benchmarks (SciMark, DaCapo) at a similar speed to the HotSpot client compiler.

2 Goals

The compiler effort aims at rewriting the high-level intermediate representation of C1X with two main goals:

Modularity: A modular design of the compiler should simplify the implementation of new languages, new back-ends, and new optimizations.

Peak Performance: A more powerful intermediate representation should enable the implementation of heavy-weight optimizations that impact the peak performance of the resulting machine code.

3 Design

For the implementation of the compiler, we rely on the following design decisions:

Graph Representation: The compiler's intermediate representation is modeled as a graph with nodes that are connected with directed edges. There is only a single node base class and every node has an associated graph object that does not change during the node's lifetime. Every node is serializable and has an id that is unique within its graph. Every edge is classified as either a control flow edge (anti-dependency) or a data flow edge (dependency) and represented as a simple pointer from the source node to the target node. It is possible to replace a node with another node without traversing the full graph. The graph does not allow data flow edge cycles or control flow edge cycles. We achieve this by explicitely modelling loops (see Section 5.2).

Extensibility: The compiler is extensible by adding new compiler phases and new node subclasses without modifying the compiler's sources. A node has an abstract way of expressing its effect and new compiler phases can ask compiler nodes for their properties and capabilities. We use the "everything is an extension" concept. Even standard compiler optimizations are internally modeled as extensions, to show that the extension mechanism exposes all necessary functionality.

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

Detailing: The compilation starts with a graph that contains nodes that represent the operations of the source language (e.g., one node for an array store to an object array). During the compilation, the nodes are replaced with more detailed nodes (e.g., the array store node is split into a null check, a bounds check, a store check, and a memory access). Compiler phases can choose whether they want to work on the earlier versions of the graph (e.g., escape analysis) or on later versions (e.g., null check elimination).

Generality: The compiler does not require Java as its input. This is achieved by having a graph as the starting point of the compilation and not a Java bytecodes array. Building the graph from the Java bytecodes must happen before giving a method to the compiler. This enables front-ends for different languages (e.g., Ruby) to provide their own graph. Also, there is no dependency on a specific back-end, but the output of the compiler is a graph that can then be converted to a different representation in a final compiler phase.

4 Milestones

The compiler is developed starting from the current C1X source code base. This helps us testing the compiler at every intermediate development step on a variety of Java benchmarks. We define the following development milestones and when they are considered achieved:

M1: We have a fully working Graal VM version with a stripped down C1X compiler that does not perform any optimizations.

M2: We modified the high-level intermediate representation to be based on the compiler graph data structure.

M3: We have reimplemented and reenabled compiler optimizations in the compiler that previously existed in C1X.

M4: We have reintegrated the new compiler into the Maxine VM and can use it as a Maxine VM bootstrapping compiler.

After those four milestones, we see three different possible further development directions that can be followed in parallel:

- Removal of the XIR template mechanism and replacement with a snippet mechanism that works with the compiler graph.
- Improvements for peak performance (loop optimizations, escape analysis, bounds check elimination, processing additional interpreter runtime feedback).
- Implementation of a prototype front-end for different languages, e.g., JavaScript.

5 Implementation

5.1 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 compiler was designed in such a way as to allow for extensive optimizations, easy traversal, compact storage and efficient processing.

5.1.1 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.

5.1.2 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 incur 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 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.
- Inlining should always be performed as a combination of two graphs.
- Nodes cannot be reassigned to another graph, they are cloned instead.

5.2 Loops

Loops form a first-class construct in the IR that is expressed in specialized IR nodes during all optimization phases. We only compile methods with a control flow where every loop has only one single entry point. This entry point is a `LoopBegin` node. This node is connected to a `LoopEnd` node that merges all control flow paths that do not exit the loop. The edge between the `LoopBegin` and the `LoopEnd` is the backedge of the loop. It goes from the beginning to the end in order to make the graph acyclic. An algorithm that traverses the control flow has to explicitely decide whether it wants to incorporate backedges (i.e., special case the treatment of `LoopEnd`) or ignore them. Figure 5.2 shows a simple example with a loop with a single entry and two exits.

5.3 Loop Phis

Data flow in loops is modelled with special phi nodes at the beginning and the end of the loop. The `LoopEnd` node merges every value that flows into the next loop iteration in associated `LoopEndPhi` nodes. A corresponding `LoopBeginPhi` node that is associated with the loop header has a control flow dependency on the `LoopEndPhi` node. Figure 5.3 shows how a simple counting loop is modelled in the graph.

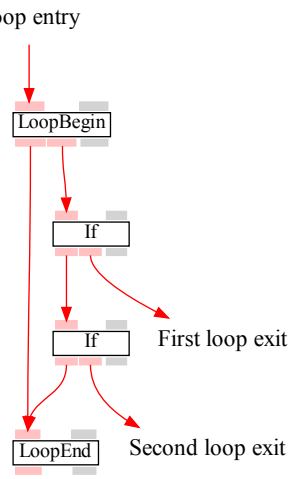


Fig. 1 A simple loop with two exits.

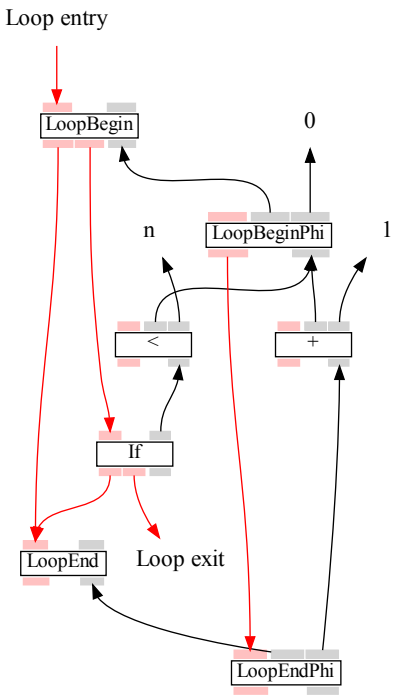


Fig. 2 Graph for a loop counting from 0 to n-1.

5.4 Loop Counters

The compiler is capable of recognizing variables that are only increased within a loop. A potential overflow of such a variable is guarded with a trap before the loop. Figure 5.4 shows the compiler graph of the example loop after the loop counter transformation.

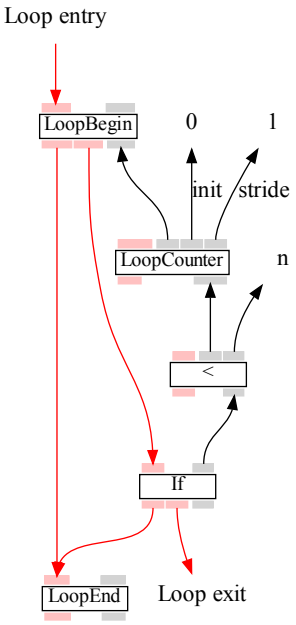


Fig. 3 Graph after loop counter transformation.

5.5 Bounded Loops

If the total maximum number of iterations of a loop is fixed, then the loop is converted into a bounded loop. The total number of iterations always denotes the number of full iterations of the loop with the control flowing from the loop begin to the loop end. If the totel number of iterations is reached, the loop is exited directly from the loop header. In the example, we can infer from the loop exit with the comparison on the loop counter that the total number of iterations of the loop is limited to n. Figure 5.5 shows the compiler graph of the example loop after the bounded loop transformation.

5.6 Vectorization

If we have now a bounded loop with no additional loop exit and no associated phi nodes (only associated loop counters), we can vectorize the loop. We replace the loop header with a normal instruction that produces

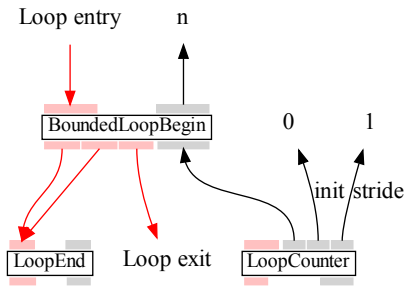


Fig. 4 Graph after bounded loop transformation.

a vector of values from 0 to the number of loop iterations minus 1. The loop counters are replaced with `VectorAdd` and `VectorMul` nodes. The vectorization is only possible if every node of the loop can be replaced with a corresponding vector node. Figure 5.6 shows the compiler graph of the example loop after vectorization. The vector nodes all work on an ordered list of integer values and are subject to canonicalization like any other node.

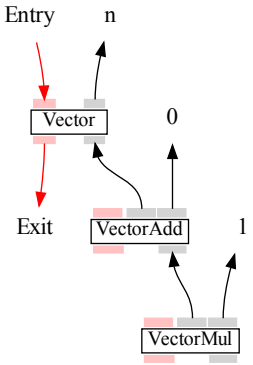


Fig. 5 Graph after bounded loop transformation.

5.7 Project Source Structure

In order to support the goal of a modular compiler, the code will be divided into the following source code projects (as subprojects of **com.oracle.graal**).

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

nodes contains the implementation of known basic nodes (e.g., phi nodes, control flow nodes, . . .). Additional node classes should go into seperate projects and be specializations of the known basic nodes.]

java contains code for building graphs from Java bytecodes and Java-specific nodes.

opt contains optimizations such as global value numbering or conditional constant propagation.

compiler contains the compiler, including:

- Schedules the compilation phases.
- Implementation of the *compiler interface* (CI).
- Implements the final compilation phase that produces the low-level representation.
- Machine code creation, including debug info.

5.8 Frame States

A frame state captures the state of the program in terms of the Java bytecode specification (i.e., the values of the local variables, the operand stack, and the locked monitors). Every deoptimization point needs a valid frame state. A frame state is valid as long as the program performs only actions that can safely be reexecuted (e.g., operations on local variables, loads, etc.). Thus, frame states need only be generated for bytecodes that cannot be reexecuted:

- Array stores: `IASTORE, LASTORE, FASTORE, DASTORE, AASTORE, BASTORE, CASTORE, SASTORE`
- Field stores: `PUTSTATIC, PUTFIELD`
- Method calls: `INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC, INVOKEINTERFACE`
- Synchronization: `MONITORENTER, MONITOREXIT`

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

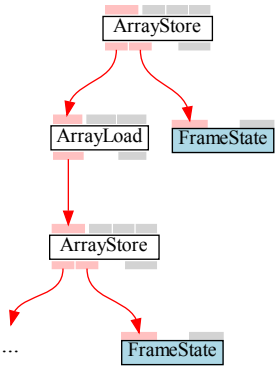


Fig. 6 Simple example using two frame states.

FrameStates also have data dependencies on the contents of the state: the local variables and the expression stack.

5.9 Traps

A trap node is a node that deoptimizes based on a conditional expression. Trap nodes are not fixed to a certain frame state node, they can move around freely and will always use the correct frame state. The node that is guarded by the deoptimization has a data dependency on the trap, and the trap in turn has a data dependency on the condition and on the most distant node that is postdominated by the guarded node.

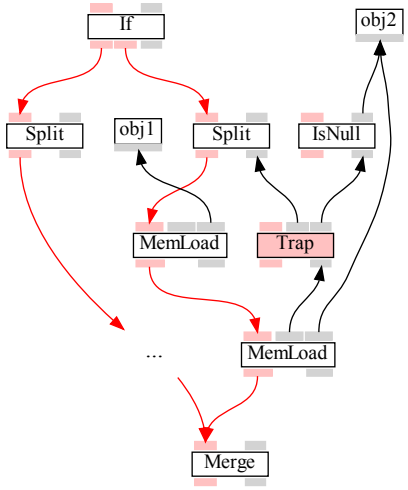


Fig. 7 In this example, the second load is guarded by a trap, because its receiver might be null (the receiver of the first load is assumed to be non-null). The trap is anchored to the control split, because as soon as this node is executed the second load must be executed as well. In the final scheduling the trap can be placed before or after the first load.

Another type of trap is a guard, which is used to remove branches that have a very low execution frequency from the compiled code.

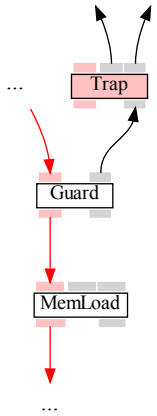


Fig. 8 In this example an If was replaced by a guard and a trap. The guard takes the place of the If in the control flow, and is connected to the trap node. The trap is now anchored to the most distant node of which the If was a postdominator.

At some point during the compilation, trap nodes need to be fixed, which means that appropriate data and control dependencies will be inserted so that they cannot move outside the scope of the associated frame state. This will generate deoptimization-free zones that can be targeted by the most aggressive optimizations. A simple algorithm for this removal of FrameStates would be to move all traps as far upwards as possible.

Multiple Traps with the same condition and anchor can be merged:

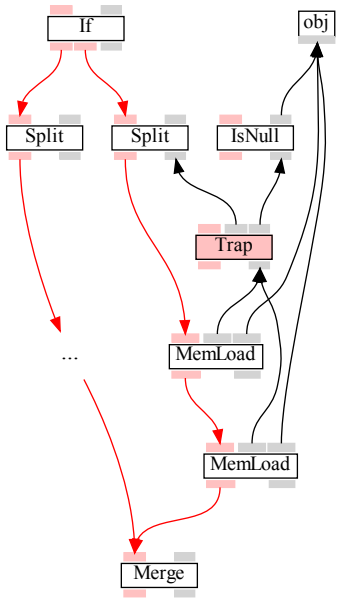
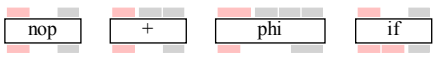


Fig. 9 Two loads guarded by the same Trap.

Also, if two Traps that are anchored to the true and false branch of the same If have the same condition, they can be merged, so that the resulting Trap is anchored at the most distant node of which the If is a postdominator.

5.10 Graphical Representation

The graphs in this document use the following node layout:



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