The Graal Compiler

Design and Strategy

work in progress

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

Created: May 4, 2011

Abstract The Graal compiler 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 Graal 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 Graal 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 Graal 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. There is no cycle in the graph that contains only control flow edges or only data flow edges. CW *What does that sentence mean? I can certainly think of a loop that has a control-flow cycle, but no data-flow cycle.* □ It is possible to replace a node with another node without traversing the full graph.

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. CW *Add: 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). CW *In general, I agree that the lowering should happen without changing the style of IR. However, I don't agree that optimizations such as null check elimination should work on a lower level graph. Isn't it bette to model "needs null check" as a capability of high-level instructions? Then the eliminator just sets a property that no null check is necessary. But that is a good discussion point: How much optimization do we want to do by augmenting a high-level IR, and how much do we want to do by rewriting a low-level IR.* □

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. CW *Here we are getting into the nits of terminology. I think the term "compiler" should always refer to the whole system that goes from bytecodes to machine code. Yes, there will be additional parsers for different bytecode formats. But still, the compiler doesn't have graphs as input and outputs, but still bytecodes and machine code, respectively.* □

4 Milestones

The Graal 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 Graal compiler graph data structure.

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

M4: We have reintegrated the new Graal 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 Graal compiler graph.
- Improvements for Graal 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 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 5.1 shows a simple example with a loop with a single entry and two exits.

5.2 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 is 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 5.2 shows how a simple counting loop is modelled in the graph.

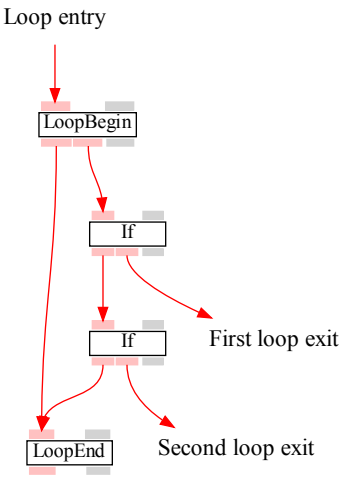


Fig. 1 A simple loop with two exits.

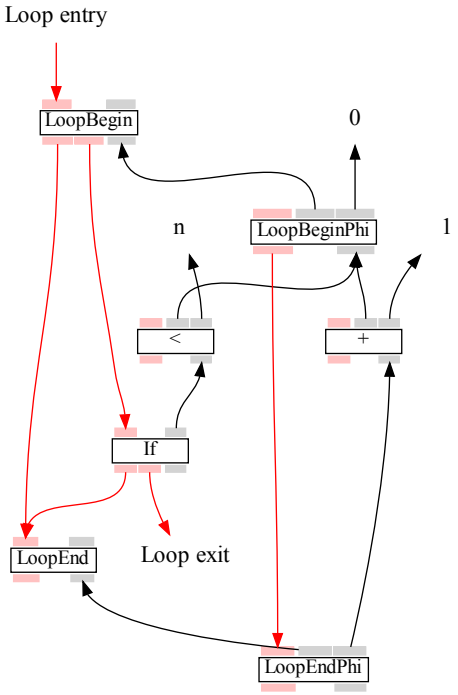


Fig. 2 Graal compiler graph for a loop counting from 0 to n-1.

5.3 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 5.3 shows the compiler graph of the example loop after the loop counter transformation.

5.4 Bounded Loops

If the total maximum number of iterations of a loop is fixed, then the loop is converted into a bounded loop.

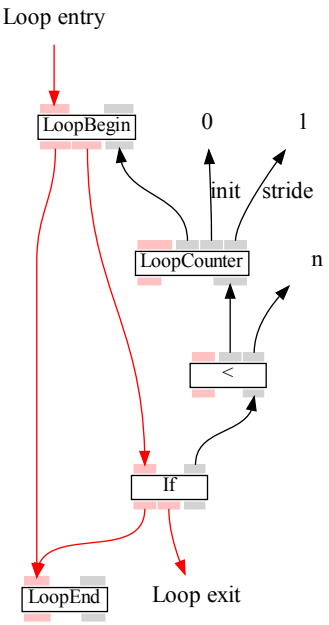


Fig. 3 Graal compiler graph after loop counter transformation.

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 5.4 shows the compiler graph of the example loop after the bounded loop transformation.

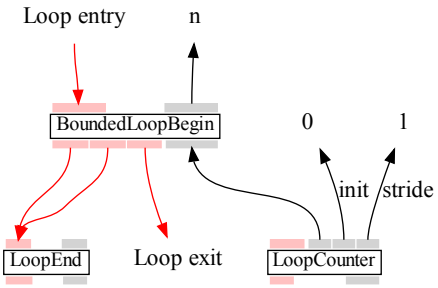


Fig. 4 Graal compiler graph after bounded loop transformation.

5.5 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

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 5.5 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.

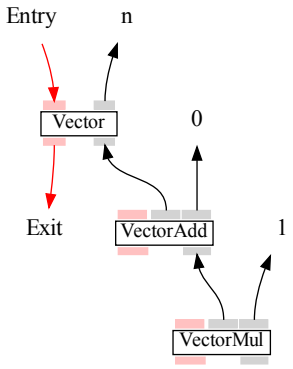


Fig. 5 Graal compiler graph after bounded loop transformation.

5.6 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: CW *Use new naming scheme com.oracle.graal...* □

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. CW *Can't we just stay with the name "instruction", which everyone understands, instead of "Node"? I strongly vote for that.* □

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.

CW *So you want to keep the backend as part of the "main compiler" at first. Seems OK for me.* □

5.7 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. CW *That cannot be an initial step, because you have nothing yet that could replace the LIR.* □

5.8 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 referred 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.

5.8.1 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. CW *The server compiler supports "renumbering" of nodes to make the ids dense again after large graph manipulations that deleted many nodes.* □
- 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). CW *I don't like that item. Cycles are a normal thing for control flow and for phi functions. I would phrase it as something like that: Nodes can only have data and control dependencies to nodes that are dominators. The only exception of that are control loop headers and phi functions* □
- 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 CW *Wrong: the client compiler only has a fixed order of pinned instructions, most instructions are not pinned and can be moved around freely* □) 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 CW *Why should there be the need for more than one graph when compiling a method?* □

5.8.2 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

5.9 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 CW *why is that an "or", both is basically the same* □ 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 cannot 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 invalidates 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.

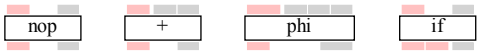
5.10 Graph Building

- 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. CW *What is that? You mean every node has x86 specific code that spits out machine code? Hope you are joking...* □ 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).
–

5.11 Graphical Representation

The graphs in this document use the following node layout:



CW *That doesn't compile with my latex. What do I have to do to get it working?* □

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