The Graal Compiler

Design and Strategy

work in progress (Oracle internal)

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

Created: June 8, 2011

Abstract The Graal compiler (simply referred to as *the compiler* in the rest of this document) aims at improving upon 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 integrated 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. This compiler-runtime interface 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 as 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 aggressive 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 explicitly modeling loops (see Section 9).

Extensibility: The compiler is extensible by allowing developers to add new compiler phases and new node subclasses without modifying the compiler's sources. A node has an abstract way of expressing its semantics 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 bytecode 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 or JavaScript) 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 being developed starting from the current C1X source code base. This helps us test the compiler at every intermediate development step on a variety of Java benchmarks. We define the following development milestones (see Section 11 for planned dates):

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

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 a different language, e.g., JavaScript.

5 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.max.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 separate 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:

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

6 Graph

The *intermediate representation* (IR) of the compiler is designed as a directed graph. The graph allocates unique ids for new nodes and can be queried for the node corresponding to a given id as well as for an unordered list of nodes of the graph. Graphs can manage side data structures (e.g. dominator trees and temporary schedules), which will be automatically invalidated and lazily recomputed whenever the graph changes. These side data structures will usually be understood by more than one optimization.

The nodes of the graph have the following properties:

- Each node is always associated with a single graph and this association is immutable.
- Each node has an immutable id that is unique within its associated graph.
- Nodes can have a data dependency, which means that one node requires the result of another 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 will be followed by the execution of another 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 in a drawing of the graph. Situations that normally incur cycles (like loops) are represented by special nodes (see Section 9).
- Ordering between nodes is specified only to the extent which is required to correctly express the semantics of a given program. This gives the compiler flexibility for the possible scheduling of a node and therefore wiggle room for 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 (see Figure 1):
  - *inputs* are all nodes that this node has data dependencies on.
  - *usages* are all nodes whose inputs contain this node.
  - *successors* are all nodes that have to be after this node in control flow.
  - *predecessors* are all nodes whose successors contain this node.
- Only inputs and successors can be changed, and changes to them will update the usages and predecessors.
- Every node must be able to support cloning and serialization.
- The edges of a node also define *happens-before* and *happens-after* relationships as shown in Figure 1.

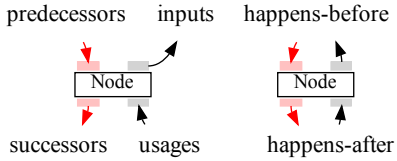

Fig. 1 A node and its edges.

6.1 Inlining

Inlining is always performed by embedding one graph into another graph. Nodes cannot be reassigned to another graph, they are cloned instead. Therefore, inlining is performed by copying and rewiring the nodes of the inlined method into the graph of the outer method. While the copying will decrease compilation performance, it enables us to cache the graph for the inlined method, optimize it independently from the outer method, and use the optimized graph for subsequent inlinings. We do not expect a significant negative impact on overall compilation performance.

We are able to perform the inlining at any point during the compilation of a method and can therefore selectively expand the inlining if a certain optimization turns out to depend on the inlining of a method. An example for this would be when the escape analysis finds out that a certain object only escapes because of one method call and this method call is not inlined, because the penalty was to high. In this case, we can chose to nevertheless inline the method in order to increase the chances for finding out that the object does not escape.

7 Control Flow

Control flow is managed in way where the predecessor node contains direct pointers to its successor nodes. We reserve the term *instruction* for nodes that are embedded in the control flow. This is opposite to the approach taken in the server compiler, where control flow and data flow edges point in the same direction. The advantage that we see in our approach is that there is no need for projection nodes in case of control flow splits. An `If` instruction can directly point to its true and false successors without any intermediate nodes. This makes the graph more compact and simplifies graph traversal.

Listing 1 shows an example Java program with an if statement where both paths do not contain any instruction with side effects. The `If` instruction can directly point its true and false successors to a `Merge` instruction. A `Phi` node that selects the appropriate value is appended to the `Merge` instruction. The `Return` instruction then has a data dependency on the `Phi` node.

```
if (condition) { return 0; }
else { return 1; }
```

Listing 1 Control flow in the graph.

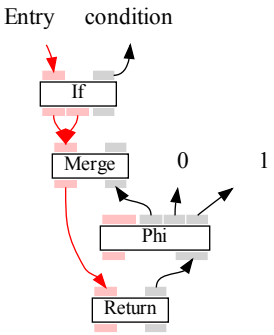

Fig. 2 A simple loop with two exits.

8 Exceptions

We do not throw runtime exceptions (e.g., `IndexOutOfBoundsException`, `NullPointerException`, or `OutOfMemoryException`), but deoptimize instead. This reduces the places in the compiled code where an exact bytecode location and debug information must be known. Additionally, this greatly reduces the number of exception handler edges in the compiled code. The main advantage of this technique is however, that we are free in moving around bounds checks, memory allocation, memory accesses with implicit null checks, etc.

There are only two kinds of instruction that need explicit exception edges, because they are the only instructions that can throw exceptions in compiled code: `Throw` instructions and `Invoke` instructions. They are modelled as instructions with an additional control flow continuation that points to an `ExceptionDispatch` instruction. The exception dispatch instruction decides based on the type of the exception object whether the control should flow to the catch handler or to another exception dispatch. If there is no catch handler in the currently compiled method, then the control flows into the `Unwind` instruction that handles the exception by forwarding it to the caller. Listing 2 shows an example Java program with nested try blocks and Figure 3 shows the corresponding compiler graph.

```
try { m1();
  try { m2();
  } catch(ExtendedException e) { ... }
  m3();
  throw exception;
} catch(Exception e) { ... }
```

Listing 2 Exception dispatch in the compiler graph.

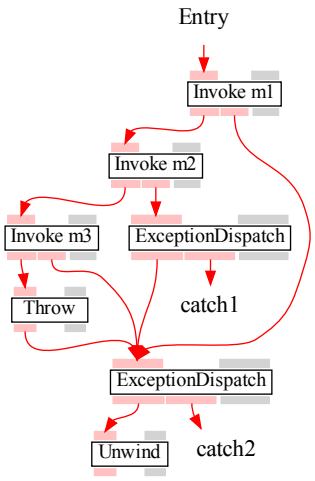

Fig. 3 A simple loop with two exits.

9 Loops

Loops form a first-class construct in the IR that is expressed by specialized IR nodes during all optimization phases. We only compile methods with a control flow where every loop has a single entry point. This entry point is a `LoopBegin` instruction. This instruction is connected to a `LoopEnd` instruction 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 of the treatment of `LoopEnd`) or ignore them. Figure 4 shows a simple example with a loop with a single entry and two exits.

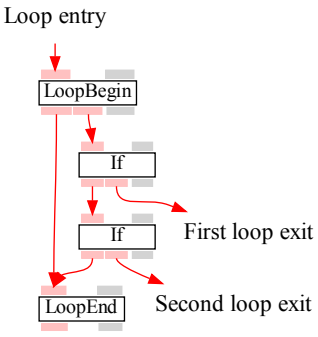

Fig. 4 A simple loop with two exits.

9.1 Loop Phis

Data flow in loops is modeled with special phi nodes at the beginning and the end of the loop. The `LoopEnd` instruction 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. Listing 3 shows a simple counting loop that is used as an example in the rest of this section. Figure 5 shows how the loop is modelled immediately after building the graph.

```
for(int i=0; i<n; ++i) { }
```

Listing 3 Loop example that counts from 0 to n-1.

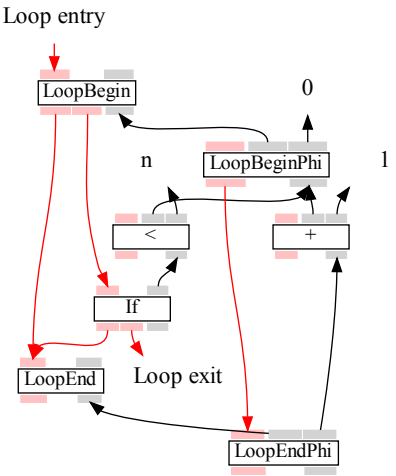

Fig. 5 Graph for a loop counting from 0 to n-1.

9.2 Loop Counters

The compiler is capable of recognizing variables that are only increased within a loop. A potential overflow of such a variable is prohibited with a guard before the loop (this is not necessary in this example, because the loop variable cannot overflow). Figure 6 shows the compiler graph of the example loop after the loop counter transformation.

9.3 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

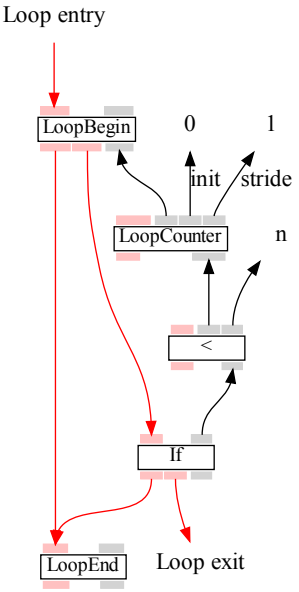

Fig. 6 Graph after loop counter transformation.

from the loop begin to the loop end. If the total 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 7 shows the compiler graph of the example loop after the bounded loop transformation.

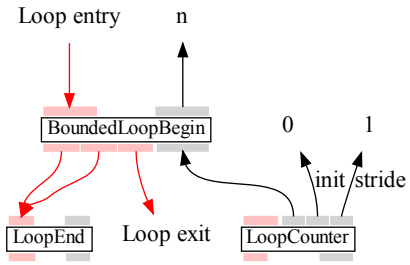

Fig. 7 Graph after bounded loop transformation.

9.4 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 8 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 and global value numbering like any other node.

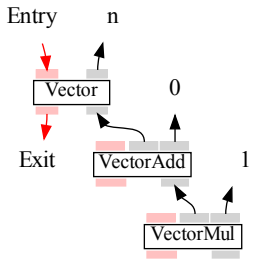

Fig. 8 Graph after vectorization.

10 Frame States

A frame state captures the state of the program like it is seen in by an interpreter of the program. The frame state contains the information that is local to the current activation and will therefore disappear during SSA-form constructions or other compiler optimizations. For Java, the frame state is defined in terms of the Java bytecode specification (i.e., the values of the local variables, the operand stack, and the locked monitors). However, a frame state is not a concept specific to Java (e.g., the Crankshaft JavaScript engine uses frame states in their optimizing compiler to model the values of the AST interpreter).

Frame states are necessary to support the deoptimization of the program, which is the precondition for performing aggressive optimizations that use optimistic assumptions. Therefore every point in the optimizing compiler that may revert execution back to the interpreter needs a valid frame state. However, the point where the interpreter continues execution need not correspond exactly to the execution position of the compiled code, because many Java bytecode instructions can be safely reexecuted. Thus, frame states need only be generated for the states after instructions that cannot be reexecuted, because they modify the state of the program. Examples for such instructions are:

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

Within the graph a frame state is represented as a node that is attached to the instruction that caused it to be generated using a control dependency (see Figure 9). Frame states also have data dependencies on the contents of the state: the local variables and the expression stack.

The frame state at the method beginning does not have to be explicitely in the graph, because it can always be reconstructed at a later stage. We save the frame state at control flow merges if there is at least one frame state on any control flow path between a node and its immediate dominator.

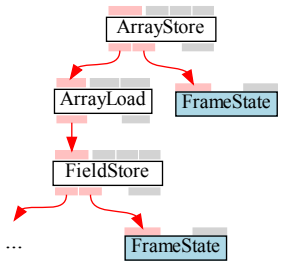

Fig. 9 Simple example using two frame states.

A deoptimization node needs a valid frame state that specifies the location and state where the interpreter should continue. The algorithm for constructing frame states makes sure that every possible location in the graph has a well-defined frame state that can be used by a deoptimization instruction. Therefore, there are no direct links between the deoptimization instruction and its frame state thus allowing the deoptimization instructions to move freely around.

10.1 Partial Escape Analysis

A partial escape analysis can help to further reduce the number of frame states. A field or array store does not create a new frame state, when the object that is modified did not have a chance to escape between its creation and the store.

Listing 4 shows an example of a method that creates two `Point` objects, connects them, and returns them. The object allocation of the first `Point` object does not need a frame state. We can always reexecute the `NEW` bytecode again in the interpreter. The `Point` object allocated by the compiler will then simply disappear after the next garbage collection. The following field store is a thread-local memory store, because the `Point` object did not have any chance to escape. Same applies to the

assignment of the `next` field and the third field assignment. Therefore, the whole method `getPoint` does not need an explicit frame state, because at any time during execution of this method, we can deoptimize and continue execution in the interpreter at the first bytecode of the method.

```
void getPoint() {
  Point p = new Point();
  p.x = 1;
  p.next = new Point();
  p.next.x = 2;
  return p;
}
```

Listing 4 Example method that needs no frame state.

The reduction of frame states makes it easier for the compiler to perform memory optimizations like memory access coalescing. We believe that this reduction on frame states is the key to effective vectorization and other compiler optimizations where compilers of compilers of unmanaged languages have advantages.

10.2 Guards

A guard is a node that deoptimizes based on a conditional expression. Guards are not attached to a certain frame state, they can move around freely and will always use the correct frame state when the nodes are scheduled (i.e., the last emitted frame state). The node that is guarded by the deoptimization has a data dependency on the guard and the guard in turn has a data dependency on the condition. A guard must not be moved above any `If` nodes. Therefore, we use `Anchor` instructions after a control flow split and a data dependency from the guard to this anchor. The anchor is the most distant instruction that is postdominated by the guarded instruction and the guard can be scheduled anywhere between those two nodes. This ensures maximum flexibility for the guard instruction and guarantees that we only deoptimize if the control flow would have reached the guarded instruction (without taking exceptions into account).

To illustrate the strengths of this approach, we show the graph for the Java code snippet shown in 5. The example looks artificial, but in case of method inlining, this is a pattern that is not unlikely to be present in a normal Java program. Figure 10 shows the compiler graph for the example method after graph building. The field stores are both represented by a single instruction and the null check that is implicitely incorporated in the field store.

```
void init(Point p) {
  if (p != null) {
    p.x = 0;
  }
  p.y = 0;
}
```

Listing 5 Example method that demonstrates the strengths of modelling the guards explicitely.

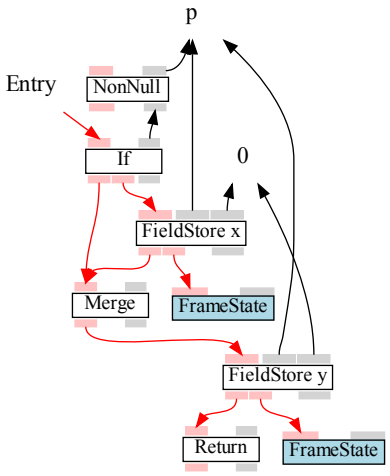

Fig. 10 Initial graph with the two field stores.

Figure 11 shows the example graph at a later compilation phase when the field store instructions are lowered to memory store instructions and explicitely modelled null check guards. The guards are attached to anchor instructions that delimit their possible schedule. The first guard must not be moved outside the `if` block; the second guard may be moved before the `If` instruction, because at this point it is already guaranteed that the second store is executed.

The first guard can be easily removed, because it is guarded by an `If` instruction that checks the same condition. Therefore we can remove the guard and the anchor from the graph and this gives us the graph shown in Figure 12.

There is another optimization for guard instructions: If two guards that are anchored to the true and false branch of the same `If` instruction have the same condition, they can be merged, so that the resulting guard is anchored at the most distant node of which the `If` instruction is a postdominator.

The remaining guard can now be moved above the `If` condition and be used to eliminate the need for the `If` node. From this point on, the guard can however no longer be moved below the first memory store. We use a control dependency from the guard to the field store to express this condition. The link between the

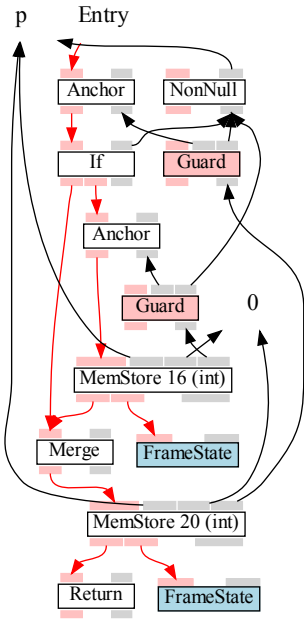

Fig. 11 A load guarded by a null check guard.

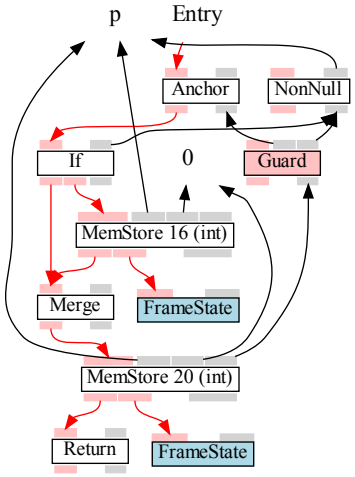

Fig. 12 After removing redundant guards.

second store and the guard and the control flow merge instruction is no longer necessary.

At some point during the compilation, guards 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 frame states would be to move all guards as far upwards as possible and then the guards are fixed using anchor nodes. In our example, the guard is already fixed, so there is no deoptimization point that uses any of the memory store

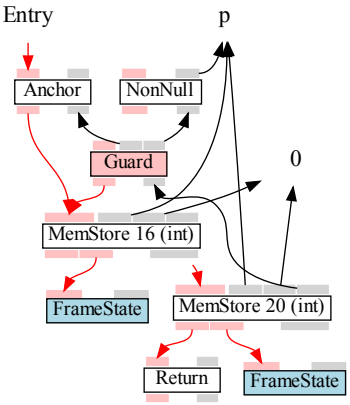

Fig. 13 After eliminating an if with a guard.

frame states. Therefore we can delete the frame states from the graph (see Figure 14).

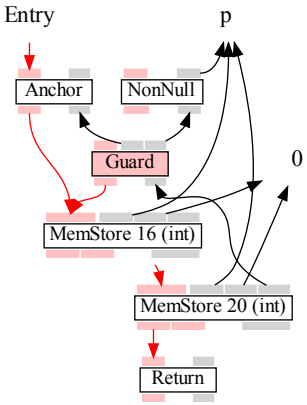

Fig. 14 After removing the frame states.

Now we can use memory coalescing to combine the two stores without frame state to adjacent locations in the same object. This is only possible if the first store does not have a frame state. Figure 15 shows the resulting graph.

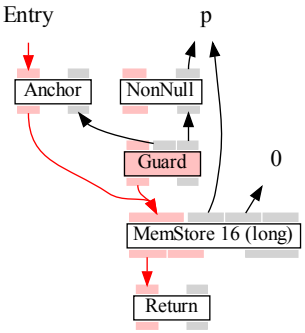

Fig. 15 After coalescing the two memory stores.

A memory store that immediately follows a null check guard instruction on the same object, can be combined into a store with an implicit null check (that deoptimizes instead of throwing the exception). Therefore, we can remove the guard again and also the anchor is no longer necessary. Figure 16 shows now that fully optimized graph that is generated for Listing 5.

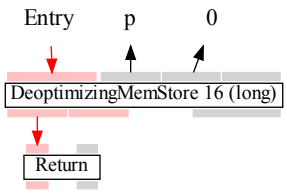

Fig. 16 Fully optimized method.

11 Conclusions

This document sketched the strategy for the Graph compiler. We already reached M1 (as defined in Section 4) and have the following plans for M2 to M4:

M2: June 30th, 2011
M3: August 15th, 2011
M4: September 30th, 2011

After we reach M4, we want to create a new project road map that further improves the Graal compiler with respect to its two main goals: Modularity and peak performance.