JOV Method Optimization

Steve Messick prepared this document early in the development of JOVE (we were still calling it JOV). This version was dated February 1998. It summarized the major optimization techniques we planned on using. Essentially all of these techniques were used in the completed versions of the compiler.

Allen Wirfs-Brock, March 2011.

JOV performs two classes of optimization. It performs intra-procedural optimizations on a single method and it performs inter-procedural optimizations on the entire program. For clarity, intra-procedural optimization is called *method optimization,* while inter-procedural optimization is called *program optimization.* This document describes the method optimizations performed on IBIC code by JOV.

The term *optimization* is a misnomer. As typically used it refers to some program transformation that is intended to improve the quality of the code in the program. Sometimes a transformation can actually make things worse, which is definitely not an optimization. Since this document describes both transformations and operations that are not, strictly speaking, transformations the more generic term *operation* is used to refer to both transformations and other analysis algorithms.

Overview of Operations

JOV performs, in order, the following operations:

- Class hierarchy analysis
- Polymorphic call-site reduction
- Procedure integration.
- Tail recursion elimination (*not* tail call optimization).
- Scalar replacement of aggregates.
- Sparse conditional constant propagation.
- Inter-procedural constant propagation (described elsewhere).
- Procedure specialization and cloning (described elsewhere).
- Sparse conditional constant propagation, again.
- Copy propagation.
- Global value numbering.
- Global code motion.
- Dead-code elimination.
- Induction-variable strength reduction.

- Linear-function test replacement.
- Induction-variable removal.
- Unnecessary bounds-checking elimination.
- Control-flow optimizations.

In addition, algebraic simplification, reassociation, and generic strength reduction are performed in a number of places. Exception set trimming must be performed after removal of code that could raise an exception. Many optimizations are described in detail in *Advanced Compiler Design and Implementation* by Steven S. Muchnick. Others are described in research reports that are referenced in the description. The reader is assumed to be familiar with the original presentation. The goal of this document is to describe the differences between JOV and other presentations.

The order is similar to that given by Muchnick. Copy propagation is moved ahead of global value numbering to simplify the latter. The following sections describe each operation. Each section has a header with three items:

- Benefit. This is a subjective estimate of how useful the operation is in improving the code produced by JOV.
- Complexity. This refers to the complexity of implementing the operation, not to the runtime complexity of the algorithm.
- Purpose. This is a short description of why the operation is included.

Benefit and complexity range from low to high. High-complexity operations require data flow analysis. None of the operations described in the original sources allow for exception handlers. If exception handlers complicate the operation that additional analysis is described in the body of the section.

Interestingly, the only need for data flow analysis is to determine live variables. Live variable analysis is needed in some induction-variable transformations and in register allocation. Chow et al [PLDI 97] point out that SSA-form is a sparse representation of the program. Data-flow algorithms typically use a dense representation that results in lots of wasted space for SSA-form. Gerlek et al [OGI tech report] present a live variable analysis algorithm based on a program representation that is related to SSA-form. We need to investigate this further; if we can use it then we will not need a traditional data-flow analyzer, nor will we need to convert out of SSA-form (which would be required? for data-flow).

Class Hierarchy Analysis

Benefit: High.
Complexity: High.
Purpose: Determine minimal type-sets of variables.

This operation is described by Chambers, Dean, and Grove in *Whole-Program Optimization of Object-Oriented Languages."* It uses data-flow analysis to determine the minimal (not necessarily minimum) set of classes that could appear in each variable of the method.

Polymorphic Call-site Reduction

Benefit: High.
Complexity: Low.
Purpose: Increase opportunities for Procedure Integration; reduce dynamic invocation.

Polymorphic call-site reduction reduces a dynamic method invocation to one or more static method invocations, with appropriate branches. This operation is not described by Muchnick. Each call site in a method is examined. Those whose dynamically-resolved method sets contain a sufficiently low number of elements are reduced to static invocations. The three instructions required to implement a dynamic invocation (method table fetching, method lookup, and dynamic invocation) are removed, provided their results are not used elsewhere. If the method set contains only one element the instruction to fetch the class of the receiver is also eliminated, and a static invocation of the method is inserted. Otherwise a series of instructions to implement inheritance test, branch on failure, static invocation, are inserted for each element of the method set. The inheritance test checks that the receiver class inherits from the implementer of the method. The branch skips to the instruction following the invocation. The test-and-branch can be eliminated for the final method of the set. Each static invocation gets the same exception handler information as the dynamic invocation had.

Initially, the call sites that qualify for this operation will have to satisfy the following constraints: the selector must not raise a checked exception, it can have no more than two implementers and, in the case of two implementers, if one of the implementing classes is a subclass of the other then the subclass must also be a leaf class.

Procedure Integration

Benefit: High.
Complexity: Low.
Purpose: Eliminate invocation overhead; provide opportunities for much greater optimization, especially with regard to loops.

Procedure integration replaces a static method invocation with the body of the method to be invoked. The basic block containing the invocation is converted to an unconditional branch block whose successor is the method entry node of the method being integrated. The method entry node is converted to an unconditional branch node containing assignments of method arguments (from the invoke statement) to method parameters. All variables in the integrated method are renamed so as not to conflict with variable names in the calling method. The method exit node of the integrated method is converted to an unconditional branch node whose successor is the previous successor of the method invocation node. Each return is converted to an unconditional branch node. If a value is returned each of these nodes contains an assignment of the return value to a new variable. (If the return value is a variable it is not necessary to create a new variable, the original can be used.) The node that replaces the method exit node contains a phi-function that selects the appropriate return variable from its predecessors an assigns it to the variable that the original invoke statement assigned to. Throw nodes in the integrated method that would cause control to transfer to an exception handler in the calling method are converted to have the exception handler as an explicit handler.

The process of determining when to apply procedure integration has not been defined yet. It is likely to be dependent on method size and loopiness. However, it will almost certainly depend on the execution profile gathered during a previous training run, when available. The profiling mechanism is described elsewhere.

Tail-recursion Elimination

Benefit: Low.
Complexity: Very low.
Purpose: Convert self-recursive methods to simple loops.

Eliminating tail recursion is very simple, except for one requirement imposed by the Java Language Specification, page 337. Invocation frames cannot be eliminated if there is a possibility that those frames could be "seen" by the program. Rather than do the full analysis required to satisfy the conditions, JOV will not eliminate tail recursion if the class SecurityManager is part of the program it is compiling.

This operation replaces recursive invocation of a method by a branch to the beginning of the method, after inserting assignments designed to have the same effect as setting method parameters to the proper value. An IBIC method begins with a series of assignments of method parameters to local variables. These assignments must be replaced with phi-functions. The first parameter to the phi-function is the method parameter. Additional parameters are added for each recursive call that is eliminated. The parameter variables are set at the point where the recursive call was made. The flow node containing the recursive call is changed to a flow node that branches to the beginning of the method. For implementation reasons, an additional node must be inserted after the first node of the program, which contains all the code previously in the first node, and is the branch target for any node branching to the beginning of the method.

Scalar Replacement of Aggregates

Purpose: Replace references to constant fields or array elements by the values held by them.
Benefit: Medium.
Complexity: Medium.

The actual effort to replace a field access expression with the value held by the field is quite small. However, determining that a field is constant and what value it holds is more difficult. Fields that are declared final can only be assigned a value in an instance initializer (for instance fields) or a class initializer (for static fields). In either case the code of the initializer must be analyzed to determine the value that is assigned to the field. In the case of final instance fields, each initializer may assign a different value, which makes this operation inapplicable.

Scalar replacement of aggregates has two phases. The first phase requires analyzing initializers, identifying updates to final fields, and determining the value that it is assigned. That value is recorded with the field, unless another value is already recorded. If so, the field is marked to identify it not being replaceable by its value (this can only happen with instance fields). The only values that will be recorded are constant primitive values, null, or string objects. If the initializer is a class initializer and the field is not

required to be present at runtime then the initialization code for the field can be removed. This is easily accomplished by removing the update instruction that stores the field value, then relying on Dead Code Removal to eliminate the remaining instructions.

The second phase applies to all methods in a class, including initializers. Each method is scanned for accesses to fields that can be replaced by their value. When found the field reference statement is removed if its value is used by nothing other than access statements having the same memory state variable. The update statement is removed and an assignment of its result variable to the replacement value is inserted in its place.

Sparse Conditional Constant Propagation

Benefit: Medium.
Complexity: Medium.
Purpose: Reduce register pressure; enable other optimizations; remove dead code.

This operation is described in Muchnick, section 12.6. The algorithm presented not only requires SSA form, it also requires that each node in the flow graph contain only one instruction. It introduces *SSA edges,* which are similar to the DU chains maintained by IBICVariable. The DU chains record the definer and a set of user instructions. The SSA edges record an edge from the definer node to one user node, with each node containing one instruction. If there are *n* users in a DU chain there will be *n* SSA edges, one for each user.

For JOV, we use a modification of Muchnick's algorithm that operates on the flow graph in its original form. The SSA edges are represented as a pair of instructions: <definer,user>. (Recall that each instructions has a reference to the basic block that contains it.) The block of code that processes FlowWL is replaced by a loop that processes each instruction in a basic block. If the first instruction in a basic block is executable (according to this algorithm) then every instruction in the block is executable. In addition, any exception handler that could be invoked as a result of executing an instruction must also be symbolically executed. An edge from the block containing the instruction to the first block of the handler is added to FlowWL.

At the completion of the algorithm we have a list of variables that contain constant values; the variables are replaced by those values and the assignment instruction is deleted. We also have a set of basic blocks that get executed; any basic block not in this set is removed. If one of the successors of a conditional branch node is deleted then that node is converted to an unconditional branch node.

One side-effect of removing code that is not discussed in Muchnick is that the set of exceptions that may be raised by a method could be reduced. This is covered in Exception Set Trimming. For this reason this operation may be most effective when applied to methods in a bottom-up traversal of the call graph.

Exception Set Trimming

Benefit: Medium.
Complexity: Low.
Purpose: Adjust the set of exceptions declared to be raised by a method.

Some IBIC instructions can cause a checked exception to be raised. When any of these instructions are removed the method must be checked to see if that exception could still be raised. We simply enumerate the instructions to see what checked exceptions are still referenced. This set then becomes the list of exceptions that a MethodDeclaration holds, which identifies the set of exceptions it can raise. Methods that invoke the modified method may contain dead exception handlers if the modified method is no longer capable of raising one or more of the exceptions it is declared to raise. Note that this operation modifies the exceptions declaration of a single method. Polymorphic call-sites need to examine the union of the exceptions declared by all methods that could be invoked at the site.

Global Value Numbering

Benefit: High.
Complexity: Medium.
Purpose: Identify duplicate expressions as preparation to Global Code Motion.

This operation is described in Muchnick, section 12.4.2, and in *Detecting Equality of Variable in Programs,* by Alpern, Wegman, and Zadeck; PoPL 1988. The algorithm presented requires SSA form. The algorithm initially constructs a partition of the value graph based on labels. Any copy-assignments cause both variables to share the same node of the value graph. (Otherwise, the claims made for Fig. 12.11.a would not hold.) Muchnick does not mention this point but the original research report does. However, doing copy propagation prior to global value numbering eliminates that concern.

Each IBIC instruction has a unique opcode. It serves as the label of internal nodes in the value graph for most instructions. Phi-assignments and method invocation require more information in their labels. The label of a phi-function includes its basic block id and number of operands. The label of a method invocation is composed of the fully-qualified class name plus the method descriptor. Leaf nodes are labeled by their value if it is a constant, or are unlabeled if it is a variable. Each node has a name in addition to its label. The name is either the variable to which the value of a node is assigned, or some arbitrary identifier if no variable is assigned that value. Copy-assignments may cause a single node to have multiple names.

At the completion of the algorithm we have a maximal partition of shared computations. Each partition containing more than one element is examined for congruent variables. Elements whose names (not labels) are variables represent congruent variables, identified by the name.

Muchnick claims that in order to use the results of Global Value Numbering we first have to determine variable equivalencies. However, our GVN output is used to drive Global Code Motion and nothing else. Since GCM is going to reschedule the instructions we can ignore equivalencies and simply substitute uses of one congruent variable for uses of all the others in the partition. For completeness, we substitute a reference to the instruction that defines the variable that is preserved for the reference to the instruction that defined the variable that is removed. This is not strictly necessary and doing so requires an additional data structure that maps from instruction to a set of duplicate references. That is necessary so that when an instruction is moved out of its basic block in final scheduling it is also moved out of other blocks that hold duplicate references. In the production

compiler we will remove this data structure but we include it during development for debugging.

In order to be able to use congruence instead of equivalence we may need to modify the GVN algorithm slightly. Click's paper makes a point of requiring certain Phi-functions that might seem removable to always be present. [I'll have to study that.]

The final result is code that has the minimum number of variables. Each value is computed once; it is associated with single variable. The ordering of instructions is invalid at the end of this operation but it will be immediately corrected by Global Code Motion.

[Alternatively, implement the GVN algorithm from Click's paper. The comparisons in his paper used the above algorithm with PRE against the GVN-GCM combo. This may be a good strategy for getting an impressive demo quickly: do Click's GVN-GCM for the Feb. demo, then consider the more precise GVN above for later. The unknown is loop detection. I was planning to use structural analysis, but it may be sufficient to follow Click's example. Muchnick describes a fairly simple loop identifier in chapter 7.]

Global Code Motion

Benefit: High.
Complexity: Medium.
Purpose: Rearrange computations to move code out of loops and into conditional branches where ever possible.

This operation is described by C. Click in *Global Code Motion, Global Value Numbering,* PLDI 1995. It replaces two operations described by Muchnick, namely Partial-Redundancy Elimination and Code Hoisting. It does not require data-flow analysis, which PRE does require, so is much easier to implement. Click's experimental results indicate that GCM produces better code that PRE more often than not. The algorithm requires the code to be in SSA form.

GCM uses the results of GVN. It is not required that GVN be run before GCM but much better code is likely to be produced if it is. GCM reschedules instructions in a three-pass operation. The first pass is to schedule early. Each instruction is moved as early in the control flow graph as possible (see the paper for details). The second pass is to schedule late. While preserving the results of the first pass, this pass moves each instruction as late in the control flow graph as possible. After finding the "minimum" and "maximum" scheduling points the third pass chooses a final schedule for each instruction that is somewhere between the minimum and maximum. The final point is chosen to move code out of loops and into the most control-dependent basic block available. The dominator tree is used to determine "early" and "late." The control dependence tree is needed to determine the most control-dependent point. A loop-finding algorithm is required to know when code is within a loop and what the loop nesting level is. The dominator tree is produced during transformation to SSA form. The control dependence tree is a minor variation on the dominator tree (see Cytron et al). Muchnick describes several algorithms that can be used to determine more or less structural information about a program, including loops. JOV uses structural analysis for this purpose. [We may want to start with

a simpler algorithm such as section 7.4 of Muchnick that simply identifies strongly-connected components by analyzing the dominator tree.]

In order to get the best effect from GCM the control flow graph must be conditioned prior to beginning the analysis. The conditioning is the same as that for PRE: insert loop pre-headers before each loop; identify and split critical edges. See Muchnick section 7.4 and 13.3, respectively, for details. This means that the dominator-dependent data structures must be recomputed prior to beginning GCM. There are incremental dominator tree management algorithms available, but they do not appear to be practical yet. In practice, it may be simplest to condition the graph prior to GVN.

Dead-Code Elimination

Benefit: Medium.
Complexity: Low.
Purpose: Eliminate code that cannot execute or does not contribute to any useful computation.

This operation is described in Muchnick, section 18.10. It is most likely to have a significant effect on code that has already been subjected to other optimizations. This is often formulated as a data-flow problem but Muchnick presents an algorithm based on DU- and UD-chains that is better suited to code in SSA-form. The algorithm is straightforward. The sets composed of block-index pairs of integers are replaced by sets of IBICInstruction instances. The initial set `Mark` that identifies the essential values is constructed by scanning the instructions for return and throw statements and adding them to `Mark`. Since DU chains are maintained by the objects that represent variables it is trivial to add users and definers of a variable to the worklist. Instructions that are marked by the algorithm will be added to a set, to indicate that they are marked.

Muchnick also describes a related operation, unreachable code elimination. Unreachable code is eliminated during control flow graph management in JOV.

Induction-Variable Strength Reduction

Benefit: Low.
Complexity: Medium.
Purpose: Replace expensive address and array expressions with cheaper calculations.

This operation is described in Muchnick, section 14.1.2. Since the full address expression needed at the machine code level is not visible to IBIC this operation has limited usefulness. It may be applicable to array-processing applications. We do not plan to implement it during initial JOV development.

Linear-Function Test Replacement & Induction-Variable Removal

Benefit: Low.
Complexity: High.
Purpose: Remove unneeded induction variables; replace loop-closing tests requiring otherwise unused independent induction variables with tests based on dependent induction variables.

This operation is described in Muchnick, section 14.1.4. It requires live variable analysis (section 14.1.3), which in turn requires data flow analysis.

Unnecessary Bounds-Checking Elimination

Benefit: Medium.
Complexity: Medium.
Purpose: Eliminate array bounds checking, or make it less expensive.

This operation is described in Muchnick, section 14.2. Global Code Motion is responsible for moving invariant bounds-checking code out of loops. This operation relies on the induction variable transformations performed by Induction-Variable Strength Reduction and Linear-Function Test Replacement. Until they are implemented this operation will not be as effective as it should be.

Control-Flow Optimizations

Muchnick describes in chapter 18 a number of control-flow transformations. Some of these, such as unreachable-code elimination and straightening, are handled as part of other operations in JOV. Others are only applicable to low-level code; branch prediction and machine idioms are examples. Unswitching, which moves loop-invariant conditional code out of loops, looks promising as a later addition to JOV.