Evaluating Value-Graph Translation Validation for LLVM

Evaluating Value-Graph Translation Validation for LLVM

Jean-Baptiste Tristan Paul Govereau Greg Morrisett
Harvard University
{tristan,govereau,greg}@seas.harvard.edu

Abstract

Translation validators are static analyzers that attempt to verify that program transformations preserve semantics. Normalizing translation validators do so by trying to match the value-graphs of an original function and its transformed counterpart. In this paper, we present the design of such a validator for LLVM's intra-procedural optimizations, a design that does not require any instrumentation of the optimizer, nor any rewriting of the source code to compile, and needs to run only once to validate a pipeline of optimizations. We present the results of our preliminary experiments on a set of benchmarks that include GCC, a perl interpreter, SQLite3, and other C programs.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages - Algebraic approaches to semantics; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs - Mechanical Verification; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs - Program and recursion schemes

General Terms Algorithms, languages, reliability, theory, verification

Keywords Translation validation, symbolic evaluation, LLVM, optimization

1. Introduction

Translation validation is a static analysis that, given two programs, tries to verify that they have the same semantics [13]. It can be used to ensure that program transformations do not introduce semantic discrepancies, or to improve testing and debugging. Previous experiments have successfully applied a range of translation validation techniques to a variety of real-world compilers. Necula [11] experimented on GCC 2.7; Zuck *et al*. [4] experimented on the Tru64 compiler; Rival [14] experimented on unoptimized GCC 3.0; and Kanade *et al*. [8] experimented on GCC 4.

Given all these results, can we effectively validate a production optimizer? For a production optimizer, we chose LLVM. To be effective, we believe our validator must satisfy the following criteria. First, we do not want to instrument the optimizer. LLVM has a large collection of program transformations that are updated and improved at a frantic pace. To be effective, we want to treat the optimizer as a "black box". Second, we do not want to modify the source code of the input programs. This means that we handle the output of the optimizer when run on the input C programs "as is." Third, we want to run only one pass of validation for the whole optimization pipeline. This is important for efficiency, and also because the boundaries between different optimizations are not always firm. Finally, it is crucial that the validator can scale to large functions. The experiments of Kanade *et al*. are the most impressive of all, with an exceptionally low rate of false alarms (note the validator requires heavy instrumentation of the compiler). However the authors admit that their approach does not scale beyond a few hundred instructions. In our experiments, we routinely have to deal with functions having several thousand instructions.

The most important issue is the instrumentation. To our knowledge, two previous works have proposed solutions that do not require instrumentation of the optimizer. Necula evaluated the effectiveness of a translation validator for GCC 2.7 with common-subexpression elimination, register allocation, scheduling, and loop inversion. The validator is simulation-based: it verifies that a simulation-relation holds between the two programs. Since the compiler is not instrumented, this simulation relation is inferred by collecting constraints. The experimental validation shows that this approach scales, as the validator handles the compilation of programs such as GCC or Linux. However, adding other optimizations such as loop-unswitch or loop-deletion is likely to break the collection of constraints.

On the other hand, Tate *et al*. [16] recently proposed a system for translation validation. They compute a value-graph for an input function and its optimized counterpart. They then augment the terms of the graphs by adding equivalent terms through a process known as *equality saturation*, resulting in a data structure similar to the E-graphs of congruence closure. If, after saturation, the two graphs are the same, they can safely conclude that the two programs they represent are equivalent. However, equality saturation was originally designed for other purposes, namely the search for better optimizations. For translation validation, it is unnecessary to saturate the value-graph, and generally more efficient and scalable to simply normalize the graph by picking an orientation to the equations that agrees with what a typical compiler will do (e.g. $1 + 1$ is replaced by 2). Their preliminary experimental evaluation for the Soot optimizer on the JVM shows that the approach is effective and can lead to an acceptable rate of false alarms. However, it is unclear how well this approach would work for the more challenging optimizations available in LLVM, such as global-value-numbering with alias analysis or sparse-conditional constant propagation.

We believe that a *normalizing* value-graph translation validator would have both the simplicity of the *saturation* validator proposed by Tate *et al*., and the scalability of Necula's constraint-based approach. In this paper we set out to evaluate how well such a design works. We therefore present the design and implementation of such a validator along with experimental results for a number of benchmarks including SQLite [2] and programs drawn from

PLDI'11, June 4–8, 2011, San Jose, California, USA.

spec marks [5], including GCC and a perl interpreter. The optimizations we consider include global-value numbering with alias analysis, sparse-conditional constant propagation, aggressive dead-code elimination, loop invariant code motion, loop unswitching, loop deletion, instruction combining, and dead-store elimination. We have also experimented, on a smaller suite of hand-written programs and optimizations, that our tool could handle without further modification various flavors of scheduling (list, trace, etc.) and loop fusion and fission.

While Tate *et al*. tried a saturation approach on JVM code and found the approach to be effective, we consider here a normalization approach in the context of C code. C code is challenging to validate because of the lack of strong typing and the relatively low-level nature of the code when compared to the JVM. However, our results show that a normalizing value-graph translation validator can effectively validate the most challenging intra-procedural optimizations of the LLVM compiler. Another significant contribution of our work is that we provide a detailed analysis of the effectiveness of the validator with respect to the different optimizations. Finally, we discuss a number of more complicated approaches that we tried but ultimately found less successful than our relatively simple architecture.

The remainder of the paper is organized as follows. Section 2 presents the design of our tool. Section 3 details the construction of the value-graph using examples. Section 4 reviews the normalization rules that we use and addresses their effectiveness through examples. Section 5 presents the results of our experiments. We discuss related work in Section 6 and conclude in Section 7.

2. Normalizing translation validation

Our validation tool is called *LLVM-MD*. At a high-level, LLVM-MD is an optimizer: it takes as input an LLVM assembly file and outputs an LLVM assembly file. The difference between our tool and the usual LLVM optimizer is that our tool certifies that the semantics of the program is preserved. LLVM-MD has two components, the usual off-the-shelf LLVM optimizer, and a translation validator. The validator takes two inputs: the assembly code of a function before and after it has been transformed by the optimizer. The validator outputs a boolean: *true* if it can prove the assembly codes have the same semantics, and *false* otherwise. Assuming the correctness of our validator, a semantics-preserving LLVM optimizer can be constructed as follows (`opt` is the command-line LLVM optimizer, `validate` is our translation validator):

```
function llvm-md(var input) {
  output = opt -options input
  for each function f in input {
    extract f from input as fi and output as fo
    if (!validate fi fo) {
      replace fo by fi in output
    }
  }
  return output
}
```

For now, our validator works on each function independently, hence the limitation to intra-procedural optimizations. We believe that standard techniques can be used to validate programs in the presence of function inlining, but have not yet implemented these. Rather, we concentrate on the workhorse intra-procedural optimizations of LLVM.

At a high level, our tool works as follows. First, it "compiles" each of the two functions into a value-graph that represents the data dependencies of the functions. Such a value-graph can be thought of as a dataflow program, or as a generalization of the result of symbolic evaluation. Then, each graph is normalized by rewriting using rules that mirror the rewritings that may be applied by the off-the-shelf optimizer. For instance, it will rewrite the 3-node sub-graph representing the expression $2+3$ into a single node representing the value 5, as this corresponds to constant folding. Finally, we compare the resulting value-graphs. If they are *syntactically* equivalent, the validator returns *true*. To make comparison efficient, the value-graphs are hash-consed (from now on, we will say "reduced"). In addition, we construct a single graph for both functions to allow sharing between the two (conceptually distinct) graphs. Therefore, in the best case—when semantics has been preserved—the comparison of the two functions has complexity $\mathcal{O}(1)$. The best-case complexity is important because we expect most optimizations to be semantics-preserving.

The LLVM-MD validation process is depicted in figure 1. First, each function is converted into Monadic Gated SSA form [6, 10, 18]. The goal of this representation is to make the assembly instructions *referentially transparent*: all of the information required to compute the value of an instruction is contained within the instruction. More importantly, referential transparency allows us to substitute sub-graphs with equivalent sub-graphs without worrying about computational effects. Computing Monadic Gated SSA form is done in two steps:

1. Make side-effects explicit in the syntax by interpreting assembly instructions as monadic commands. For example, a `load` instruction will have a extra parameter representing the memory state.
2. Compute Gated SSA form, which extends ϕ-nodes with conditions, insert μ-nodes at loop headers, and η-nodes at loop exits[18].

Once we have the Monadic Gated SSA form, we compute a shared value-graph by replacing each variable with its definition, being careful to maximize sharing within the graph. Finally, we apply normalization rules and maximize sharing until the value of the two functions merge into a single node, or we cannot perform any more normalization.

It is important to note that the precision of the semantics-preservation property depends on the precision of the monadic form. If the monadic representation does not model arithmetic overflow or exceptions, then a successful validation does not guarantee anything about those effects. At present, we model memory state, including the local stack frame and the heap. We do not model runtime errors or non-termination, although our approach can be extended to include them. Hence, a successful run of our validator implies that if the input function terminates and does not produce a runtime error, then the output function has the same semantics. Our tool does not yet offer *formal* guarantees for non-terminating or semantically undefined programs.

3. Validation by Example

3.1 Basic Blocks

We begin by explaining how the validation process works for basic blocks. Considering basic blocks is interesting because it allows us to focus on the monadic representation and the construction of the value-graph, leaving for later the tricky problem of placing gates in ϕ- and η-nodes.

Our validator uses LLVM assembly language as input. LLVM is a portable SSA-form assembly language with an infinite number of registers. Because we start with SSA-form, producing the value-graph consists of replacing variables by their definitions while maximizing sharing among graph nodes. For example, consider the

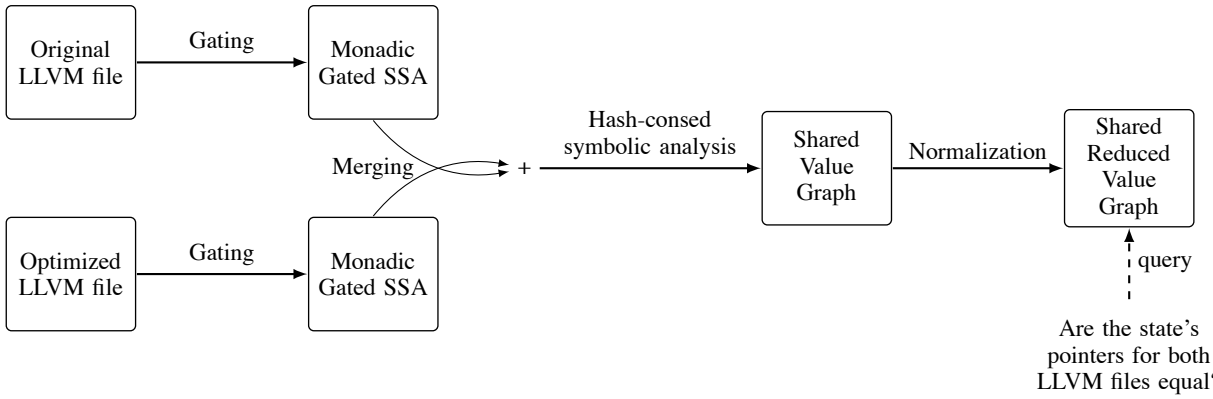


Figure 1: LLVM M.D. from a bird's eye view

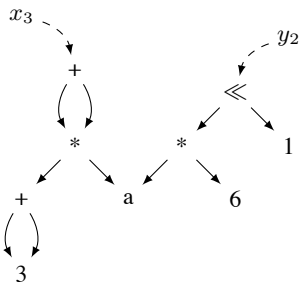
following basic block, B1:

$$\begin{aligned} \text{B1:}\quad & x_1 = 3 + 3 \\ & x_2 = a * x_1 \\ & x_3 = x_2 + x_2 \end{aligned}$$

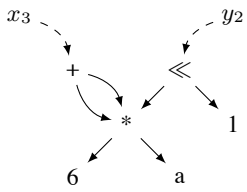
and its optimized counterpart, B2:

$$\begin{aligned} \text{B2:}\quad & y_1 = a * 6 \\ & y_2 = y_1 << 1 \end{aligned}$$

Replacing variables x_1, x_2, and y_1 by their definition, we obtain the value-graph presented below. The dashed arrows are not part of the value graph, they are only meant to point out which parts of the graph correspond to which program variables. Note that both blocks have been represented within one value graph, and the node for the variable `a` has been shared.



Suppose that we want to show that the variables x_3 and y_2 will hold the same value. Once we have the shared value graph in hand, we simply need to check if x_3 and y_2 are represented by subgraphs rooted at the same graph node. In the value graph above, x_3 and y_2 are not represented by the same subgraph, so we cannot conclude they are equivalent. However, we can now apply normalization rules to the graph. First, we can apply a constant folding rule to reduce the subgraph $3 + 3$ to a single node 6. The resulting graph is shown below (we have maximized sharing in the graph).



We have managed to make the value graph smaller, but we still cannot conclude that the two variables are equivalent. So, we continue to normalize the graph. A second rewrite rule allows us to replace $x + x$ with $x \ll 1$ for any x. In our setting, this rule is only appropriate if the optimizer would prefer the shift instruction to addition (which LLVM does). After replacing addition with left shift, and maximizing sharing, x_3 and y_2 point to the same node and we can conclude that the two blocks are equivalent.

Side Effects. The translation we have described up to this point would not be correct in the presence of side effects. Consider the following basic block.

$$\begin{aligned} & p_1 = \texttt{alloc}\ 1 \\ & p_2 = \texttt{alloc}\ 1 \\ & \texttt{store}\ x, p_1 \\ & \texttt{store}\ y, p_2 \\ & z = \texttt{load}\ p_1 \end{aligned}$$

If we naively apply our translation, then the graph corresponding to z would be: $z \mapsto \texttt{load}\ (\texttt{alloc}\ 1)$, which does not capture the complete computation for register z. In order to make sure that we do not lose track of side effects, we use abstract state variables to capture the dependencies between instructions. A simple translation gives the following sequence of instructions for this block:

$$\begin{aligned} p_1, m_1 &= \texttt{alloc}\ 1, m_0 \\ p_2, m_2 &= \texttt{alloc}\ 1, m_1 \\ m_3 &= \texttt{store}\ x, p_1, m_2 \\ m_4 &= \texttt{store}\ y, p_2, m_3 \\ z, m_5 &= \texttt{load}\ p_1, m_4 \end{aligned}$$

Here, the current memory state is represented by the m registers. Each instruction requires and produces a memory register in addition to its usual parameters. This extra register enforces a dependency between, for instance, the `load` instruction and the preceding `store` instructions. This translation is the same as we would get if we interpreted the assembly instructions as a sequence of monadic commands in a simple state monad[10]. Using these "monadic" instructions, we can apply our transformation and produce a value graph that captures all of the relevant information for each register.

The rewriting rules in our system are able to take into account aliasing information to relax the strict ordering of instructions imposed by the monadic transformation. In our setting (LLVM), we know that pointers returned by `alloc` never alias with each other.

Using this information, we are able to replace m_4 with m_3 in the `load` instruction for z. Then, because we have a `load` from a memory consisting of a `store` to the same pointer, we can simplify the `load` to x.

Using the state variables, we can validate that a function not only computes the same value as another function, but also affects the heap in the same way. The same technique can be applied to different kinds of side effects, such as arithmetic overflow, division by zero, and non-termination. Thus far we have only modeled memory side-effects in our implementation. Hence, we only prove semantics preservation for terminating programs that do not raise runtime errors. However, our structure allows us to easily extend our implementation to a more accurate model, though doing so may make it harder to validate optimizations.

3.2 Extended Basic Blocks

These ideas can be generalized to extended basic blocks as long as ϕ-nodes are referentially transparent. We ensure this through the use of *gated* ϕ-nodes. Consider the following program, which uses a normal ϕ-node as you would find in an SSA-form assembly program.

$$\begin{array}{lll}
\textbf{entry}: & c = a < b & \\
& \texttt{cbr}\ c, \textbf{true}, \textbf{false} & \\
\\
\textbf{true}: & x_1 = x_0 + x_0 & \text{(True branch)} \\
& \texttt{br}\ \textbf{join} & \\
\\
\textbf{false}: & x_2 = x_0 * x_0 & \text{(False branch)} \\
& \texttt{br}\ \textbf{join} & \\
\\
\textbf{join}: & x_3 = \phi(x_1, x_2) & \text{(Join point)}
\end{array}$$

Rewriting these instructions as is would not be correct. For instance, replacing the condition $a < b$ by $a \geq b$ would result in the same value-graph, and we could not distinguish these two different programs. However, if the ϕ-node is extended to include the conditions for taking each branch, then we can distinguish the two programs. In our example, the last instruction would become $x_3 = \phi(b, x_1, x_2)$, which means x_3 is x_1 if b is *true*, and x_2 otherwise.

In order to handle real C programs, the actual syntax of ϕ-nodes has to be a bit more complex. In general, a ϕ-node is composed of a set of possible branches, one for each control-flow edge that enters the ϕ-node. Each branch has a set of conditions, all of which must be true for the branch to be taken.[1]

$$\phi \left\{ \begin{array}{c} c_{11} \ldots c_{1n} \rightarrow v_1 \\ \cdots \\ c_{k1} \ldots c_{km} \rightarrow v_k \end{array} \right\}$$

Given this more general syntax, the notation $\phi(c, x, y)$ is simply shorthand for $\phi(c \rightarrow x, !c \rightarrow y)$.

Gated ϕ nodes come along with a set of normalization rules that we present in the next section. When generating gated ϕ-nodes, it is important that the conditions for each branch are mutually exclusive with the other branches. This way, we are free to apply normalization rules to ϕ-nodes (such as reordering conditions and branches) without worrying about changing the semantics.

It is also worth noting that if the values coming from various paths are equivalent, then they will be shared in the value graph. This makes it possible to validate optimizations based on global-value numbering that is aware of equivalences between definitions from distinct paths.

[1] ϕ-nodes with several branches and many conditions are very common in C programs. For example, an if-statement with a condition that uses short-cut boolean operators, can produce complex ϕ-nodes.

3.3 Loops

In order to generalize our technique to loops, we must come up with a way to place gates within looping control flow (including breaks, continues, and returns from within a loop). Also, we need a way to represent values constructed within loops in a referentially transparent way. Happily, Gated SSA form is ideally suited to our purpose.

Gated SSA uses two constructs to represent loops in a referentially transparent fashion. The first, μ, is used to define variables that are modified within a loop. Each μ is placed at a loop header and holds the initial value of the variable on entry to the loop and a value for successive iterations. The μ-node is equivalent to a non-gated ϕ node from classical SSA. The second, η, is used to refer to loop-defined variables from outside their defining loops. The η-node carries the variable being referred to along with the condition required to reach the η from the variable definition.

Figure 2a shows a simple while loop in SSA form. The Gated SSA form of the same loop is shown in figure 2b. The ϕ-nodes in the loop header have been replaced with μ-nodes, and the access to the x_p register from outside to loop is transformed into an η-node that carries the condition required to exit the loop and arrive at this definition.

With Gated SSA form, recursively defined variables must contain a μ-node. The recursion can be thought of as a cycle in the value graph, and all cycles are dominated by a μ-node. The value graph corresponding to our previous example is presented in figure 2c. Intuitively, the cycle in the value-graph can be thought of as generating a stream of values. The μ-nodes start by selecting the initial value from the arrow marked with an "i". Successive values are generated from the cyclic graph structure attached to the other arrow. This μ-node "produces" the values c, $c+1$, $c+2$,... The η receives a stream of values and a stream of conditions. When the stream of conditions goes from *true* to *false*, the η selects the corresponding value in the value stream.

Generally, we can think of μ and η behaving according to the following formulas:

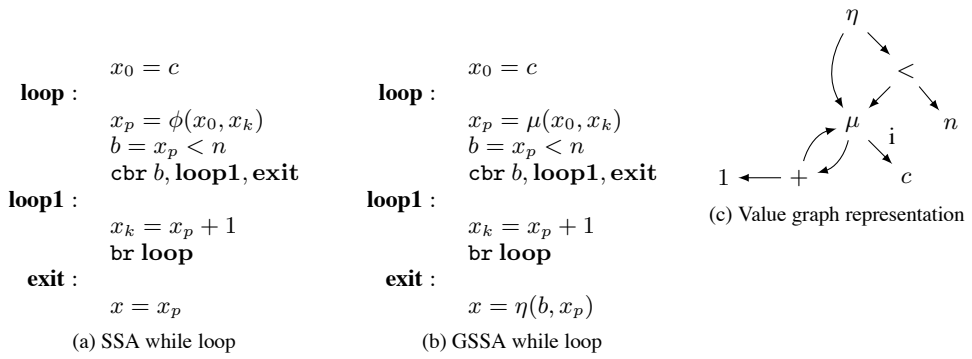
$$\begin{aligned}
\mu(a, n) &= a : \mu(n[a/x], n) \\
\eta(0 : \overline{b}, x : \overline{v}) &= \eta(\overline{b}, \overline{v}) \\
\eta(1 : \overline{b}, x : \overline{v}) &= x
\end{aligned}$$

Of course, for our purposes, we do not need to evaluate these formulas, we simply need an adequate, symbolic representation for the registers x, x_p and b_p. A more formal semantics should probably borrow ideas from dataflow programming languages such as those studied by Tate *et. al.*[16].

4. Normalization

Once a graph is constructed for two functions, if the functions' values are not already equivalent, we begin to normalize the graph. We normalize value graphs using a set of rewrite rules. We apply the rules to each graph node individually. When no more rules can be applied, we maximize sharing within the graph and then reapply our rules. When no more sharing or rules can be applied, the process terminates.

Our rewrite rules come in two basic types: general simplification rules and optimization-specific rules. The general simplification rules reduce the number of graph nodes by removing unnecessary structure. We say *general* because these rules only depend on the graph representation, replacing graph structures with smaller, simpler graph structures. The optimization-specific rules rewrite graphs in a way that mirrors the effects of specific optimizations.



(a) SSA while loop (b) GSSA while loop (c) Value graph representation

Figure 2: Representation of while loops

These rules do not always make the graph smaller or simpler, and one often needs to have specific optimizations in mind when adding them to the system.

General Simplification Rules. The notation $a \downarrow b$ means that we match graphs with structure a and replace them with b. The first four general ruless simplify boolean expressions:

$$a = a \downarrow \texttt{true} \tag{1}$$
$$a \neq a \downarrow \texttt{false} \tag{2}$$
$$a = \texttt{true} \downarrow a \tag{3}$$
$$a \neq \texttt{false} \downarrow a \tag{4}$$

These last two rules only apply if the comparison is performed at the boolean type. In fact, all LLVM operations, and hence our graph nodes, are typed. The types of operators are important and uninteresting: we do not discuss types in this paper.

There are two general rules for removing unnecessary ϕ-nodes.

$$\phi\{\ldots, \overline{\texttt{true}_i} \to t, \ldots\} \downarrow t \tag{5}$$
$$\phi\left\{\overline{c_i \to t}\right\} \downarrow t \tag{6}$$

The first rule replaces a ϕ-node with one of its branches if all of its conditions are satisfied for that branch. We write $\overline{x_i}$ for a set of terms indexed by i. In the first rule, we have a set of true values. Note that the conditions for each branch are mutually exclusive with the other branches, so only one branch can have conditions which are all true. The second rule removes the ϕ-node if all of the branches contain the same value. A special case of this rule is a ϕ-node with only one branch indicating that there is only one possible path to the ϕ-node, as happens with branch elimination.

The ϕ rules are required to validate sparse conditional constant propagation (SCCP) and global value numbering (GVN). The following example can be optimized by both:

```
if (c) {a = 1; b = 1; d = a;}
  else {a = 2; b = 2; d = 1;}
if (a == b) {x = d;} else {x = 0;}
return x;
```

Applying global-value numbering followed by sparse conditional constant propagation transforms this program to `return 1`. Indeed, in each of the branches of the first if-statement, a is equal to b. Since $a == b$ is always true, the condition of the second if-statement is constant, and sparse conditional constant propagation can propagate the left definition of x. The above program and `return 1` have the same normalized value graph, computed as follows:

$$\begin{aligned} x \mapsto & \phi(\phi(c,1,2) == \phi(c,1,2), \phi(c,1,1), 0) \\ & \downarrow \phi(\texttt{true}, \phi(c,1,1), 0) && \text{by (1)} \\ & \downarrow \phi(c,1,1) && \text{by (5)} \\ & \downarrow 1 && \text{by (6)} \end{aligned}$$

There are also general rules for simplifying η- and μ-nodes. The first rule allows us to remove loops that never execute.

$$\eta(\texttt{false}, \mu(x,y)) \downarrow x \tag{7}$$

This rule rewrites to the initial value of the μ-node before the loop is entered, namely x. This rule is needed to validate loop-deletion, a form of dead code elimination. In addition, there are two rules for loop invariants. The first says that if we have a constant μ-node, then the corresponding η-node can be removed:

$$\eta(c, \mu(x,x)) \downarrow x \tag{8}$$
$$\eta(c, y \mapsto \mu(x,y)) \downarrow x \tag{9}$$

In rule (8), the μ-node has an initial value of x, which must be defined outside of the loop, and therefore cannot vary within the loop. Since, x does not vary within the loop the μ-node does not vary, and the loop structure can be removed. Rule (9), expresses the same condition, but the second term in the μ-node is again the same μ-node (we use the notation $y \mapsto \mu(x,y)$ to represent this self-reference).

These rules are necessary to validate loop invariant code motion. As an example, consider the following program:

```
x = a + 3; c = 3;
for (i = 0; i < n; i++) {x = a + c;}
return x;
```

In this program, variable x is defined within a loop, but it is invariant. Moreover, variable c is a constant. Applying global constant propagation, followed by loop-invariant code motion and loop deletion transforms the program to `return (a + 3)`. The value graph for x is computed as follows:

$$\begin{aligned} & i_n \mapsto \mu(0, i_n + 1) \\ & x \mapsto \eta(i_n < n, \mu(a+3, a+3)) \\ & \quad \downarrow a + c && \text{(by 8)} \end{aligned}$$

Note that the global copy propagation is taken care of "automatically" by our representation, and we can apply our rule (8) immediately. The other nodes of the graph (i_n) are eliminated since they are no longer needed.

Optimization-specific Rules. In addition to the general rules, we also have a number of rewrite rules that are derived from the semantics of LLVM. For example, we have a family of laws for simplifying constant expressions, such as:

$$\texttt{add}\ 3\ 2 \downarrow 5$$
$$\texttt{mul}\ 3\ 2 \downarrow 6$$
$$\texttt{sub}\ 3\ 2 \downarrow 1$$

Currently we have rules for simplifying constant expressions over integers, but not floating point or vector types. There are also rules for rewriting instructions such as:

$$\texttt{add}\ a\ a \downarrow \texttt{shl}\ a\ 1$$
$$\texttt{mul}\ a\ 4 \downarrow \texttt{shl}\ a\ 2$$

These last two rules are included in our validator because we know the LLVM's optimizer prefers the shift left instruction. While preferring shift left may be obvious, there are some less obvious rules such as:

$$\texttt{add}\ x\ (-k) \downarrow \texttt{sub}\ x\ k$$
$$\texttt{gt}\ 10\ a \downarrow \texttt{lt}\ a\ 10$$
$$\texttt{lt}\ a\ b \downarrow \texttt{le}\ a\ (\texttt{sub}\ b\ 1)$$

While these transformations may not be optimizations, we believe LLVM makes them to give the instructions a more regular structure.

Finally, we also have rules that make use of aliasing information to simplify memory accesses. For example, we have the following two rules for simplifying loads from memory:

$$\texttt{load}(p, \texttt{store}(x, q, m)) \downarrow \texttt{load}(p, m) \quad (10)$$
$$\texttt{load}(p, \texttt{store}(x, p, m)) \downarrow x \quad (11)$$

when p and q do not alias. Our validator can use the result of a may alias analysis. For now, we only use simple non-aliasing rules: two pointers that originate from two distinct stack allocations may not alias; two pointers forged using `getelemptr` with different parameters may not alias, etc.

4.1 Efficiency

At this point is natural to wonder why we did not simply define a normal form for expressions and rewrite our graphs to this normal form. This is a good option, and we have experimented with this strategy using external SMT provers to find equivalences between expressions. However, one of our goals is to build a practical tool which optimizes the best case performance of the validator (we expect most optimizations to be correct). Using our strategy of performing rewrites motivated by the optimizer, we are often able to validate functions with tens of thousands of instructions (resulting in value graphs with hundreds of thousands of nodes) with only a few dozen rewritings. That is, we strive to make the amount of work done by the validator proportional to the number of transformations performed by the optimizer.

To this end, the rewrite rules derived from LLVM semantics are designed to mirror the kinds of rewritings that are done by the LLVM optimization pipeline. In practice, it is *much* more efficient to transform the value graphs in the same way the optimizer transforms the assembly code: if we know that LLVM will prefer `shl` a 1 to $a + a$, then we will rewrite $a + a$ but not the other way around.

As another, more extreme, example, consider the following C code, and two possible optimizations: SCCP and GVN.

```
a = x < y;
b = x < y;
if (a) {
   if (a == b) {c = 1;} else {c = 2;}
} else {c = 1;}
return c;
```

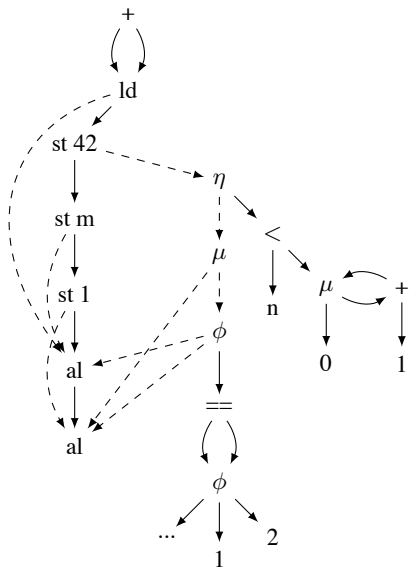


Figure 3: A shared value-graph

If the optimization pipeline is setup to apply SCCP first, then a may be replaced by `true`. In this case, GVN can not recognize that a and b are equal, and the inner condition will not be simplified. However, if GVN is applied first, then the inner condition can be simplified, and SCCP will propagate the value of c, leading to the program that simply returns 1. The problem of how to order optimizations is well-known, and an optimization pipeline may be reordered to achieve better results. If the optimization pipeline is configured to use GVN before SCCP, then, for efficiency, our simplification should be setup to simplify at join points before we substitute the value of a.

4.2 Extended Example

We now present an example where all these laws interplay to produce the normalized value-graph. Consider the C code below:

```
int f(int n, int m) {
  int * t = NULL;
  int * t1 = alloca(sizeof(int));
  int * t2 = alloca(sizeof(int));
  int x, y, z = 0;
  *t1 = 1; *t2 = m;
  t = t1;
  for (int i = 0; i < n; ++i) {
    if (i % 3) {
      x = 1; z = x << y; y = x;
    } else {
      x = 2; y = 2;
    }
    if (x == y)  t = t1;
    else t = t2;
  }
  *t = 42;
  return *t2 + *t2;
}
```

First, note that this function returns $m + m$. Indeed, x is always equal to y after the execution of the first conditional statement in

the for loop. Therefore, the second conditional statement always executes the left branch and assigns t_1 to t. This is actually a loop invariant, and, since t_1 is assigned to t before the loop, t_1 is always equal to t. t_1 and t_2 are pointers to two distinct regions of memory and cannot alias. Writing through t_1 does not affect what is pointed to by t_2, namely m. The function therefore returns $m + m$. Since the loop terminates, an optimizer may replace the body of this function with $m \ll 1$, using a blend of global-value numbering with alias analysis, sparse-conditional constant propagation and loop deletion.

Our value-graph construction and normalization produces the value-graph corresponding to $m \ll 1$ for this example. The initial value-graph is presented in figure 3. Some details of the graph have been elided for clarity. We represent `load`, `store`, and `alloca` nodes with `ld`, `st`, and `al` respectively. To make the graph easier to read, we also used dashed lines for edges that go to pointer values. Below is a potential normalization scenario.

- The arguments of the $==$ node are shared; it is rewritten to *true*.
- The gate of the ϕ node is *true*; its predecessor, μ, is modified to point to the target of the true branch of the ϕ node rather than to ϕ itself.
- The arguments of this μ node are shared; its predecessor, η, is modified to point to the target of μ instead of μ itself.
- The condition of the η node is terminating, and its value acyclic; Its predecessor, `st`42, is modified to point to the target of η instead of η itself.
- It is now obvious that the load and its following store use distinct memory regions; the load can "jump over the store."
- The load and its new store use the same memory region; the load is therefore replaced by the stored value, m.
- The arguments of the $+$ node are shared; The $+$ node is rewritten into a left shift.

5. Experimental evaluation

In our experimental evaluation, we aim to answer three different questions.

1. How effective is the tool for a decent pipeline of optimization?
2. How effective is the tool optimization-by-optimization?
3. What is the effect of normalization?

Our measure of effectiveness is simple: the fewer false alarms our tool produces, the more optimized the code will be. Put another way, assuming the optimizer is always correct, what is the cost of validation? In our experiments, we consider any alarm to be a false alarm.

5.1 The big picture

We have tested our prototype validator on the pure C programs of the SPECCPU 2006[5] benchmark (The xalancbmk benchmark is missing because the LLVM bitcode linker fails on this large program). In addition, we also tested our validator on the SQLite embedded database[2]. Each program was first compiled with clang version 2.8[1], and then processed with the mem2reg pass of the LLVM compiler to place ϕ-nodes. These assembly files make up the unoptimized inputs to our validation tool. Each unoptimized input was optimized with LLVM, and the result compared to the unoptimized input by our validation tool.

Test suite information. Table 1 lists the benchmark programs with the size of the LLVM-assembly code file, number of lines of assembly, and the number of functions in the program. Our research prototype does not analyze functions with irreducible control flow graphs. This restriction comes from the front-end's computation of Gated SSA form. It is well known how to compute Gated SSA form for any control-flow graph[18]. However, we have not extended our front-end to handle irreducible control flow graphs. We do not believe the few irreducible functions in our benchmarks will pose any new problems, since neither the Gated SSA form nor our graph representation would need to be modified.

	size	LOC	functions
SQLite	5.6M	136K	1363
bzip2	904K	23K	104
gcc	63M	1.48M	5745
h264ref	7.3M	190K	610
hmmer	3.3M	90K	644
lbm	161K	5K	19
libquantum	337K	9K	115
mcf	149K	3K	24
milc	1.2M	32K	237
perlbench	15M	399K	1998
sjeng	1.5M	39K	166
sphinx	1.7M	44K	391

Table 1: Test suite information

Pipeline information. For our experiment, we used a pipeline consisting of:

- ADCE (advanced dead code elimination), followed by
- GVN (global value numbering),
- SCCP (sparse-condition constant propagation),
- LICM (loop invariant code motion),
- LD (loop deletion),
- LU (loop unswitching),
- DSE (dead store elimination).

The optimizations we chose are meant to cover, as much as possible, the intra-procedural optimizations available in LLVM. We do not include constant propagation and constant folding because both of these are subsumed by sparse-conditional constant propagation (SCCP). Similarly, dead-code and dead-instruction elimination are subsumed by aggressive dead-code elimination (ADCE). We did not include reassociate and instcombine because we haven't addressed those yet. One reason these haven't been our priority is that they are conceptually simple to validate but require many rules.

For each benchmark, we ran all of the optimizations, and then attempted to validate the final result. Since we used the SQLite benchmark to engineer our rules, it is not surprising that, overall, that benchmark is very close to 90%. The rules chosen by studying SQLite are also very effective across the other benchmarks. We do not do quite as well for the perlbench and gcc benchmarks. Currently, our tool only uses the basic rewrite rules we have described here. Also we do not handle global constants or floating point expressions.

Pipeline Results. Overall, with the small number of rules we have described in this paper, we can validate 80% of the per-function optimizations. The results per-benchmark are shown in figure 4. For our measurements, we counted the number of functions for which we could validate all of the optimizations performed on the function: even though we may validate many optimizations, if even one optimization fails to validate we count the entire function as failed. We found that this conservative approach, while rejecting

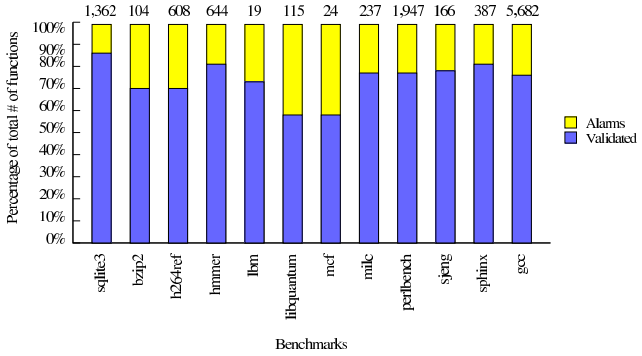


Figure 4: Validation results for optimization pipeline

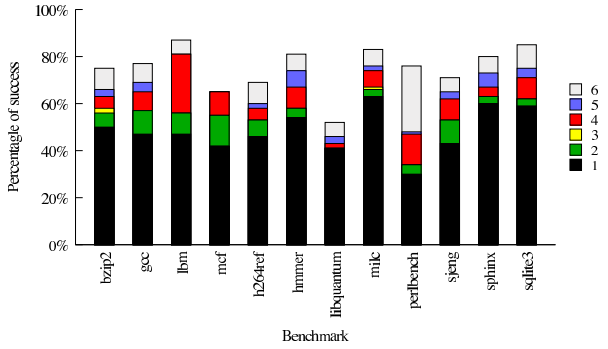


Figure 6: GVN

more optimizations, leads to a simpler design for our validated optimizer which rejects or accepts whole functions at a time. The validation time for GCC is 19m19s, perl 2m56s, and SQLite 55s.

5.2 Testing Individual Optimizations

The charts in Figure 5 summarize the results of validating functions for different, single optimizations. The height of each bar indicates the total number of functions transformed for a given benchmark and optimization. The bar is split showing the number of validated (below) and unvalidated (above) functions. The total number of functions is lower in these charts because we do not count functions that are not transformed by the optimization.

It is clear from the charts that GVN with alias analysis is the most challenging optimization for our tool. It is also the most important as it performs many more transformations than the other optimizations. In the next section we will study the effectiveness of rewrite rules on our system.

5.3 Rewrite Rules

Figure 6 shows the effect of different rewrite rules for the GVN optimization with our benchmarks. The total height of each bar shows the percentage of functions we validated for each benchmark after the GVN optimization. The bars are divided to show how the results improve as we add rewrite rules to the system. We start with no rewrite rules, then we add rules and measure the improvement. The bars correspond to adding rules as follows:

1. no rules
2. ϕ simplification
3. constant folding
4. load/store simplification
5. η simplification
6. commuting rules

We have already described the first five rules. The last set of rules tries to rearrange the graph nodes to enable the former rules. For example, we have a rule that tries to "push down" η-nodes to get them close to the matching μ-nodes.

We can see from this chart that different benchmarks are effected differently by the different rules. For example, SQLite is not improved by adding rules for constant folding or ϕ simplification. However, load/store simplification has an effect. This is probably because SQLite has been carefully tuned by hand and does not have many opportunities for constant folding or branch elimination. The lbm benchmark, on the other hand, benefits quite a lot from ϕ simplification.

From the data we have, it seems that our technique is able to successfully validate approximately 50% of GVN optimizations with no rewrite rules at all. This makes intuitive sense because our symbolic evaluation hides many of the syntactic details of the programs, and the transformations performed by many optimizations are, in the end, minor syntactic changes. By adding rewrite rules we can dramatically improve our results.

Up to this point, we have avoided adding "special purpose" rules. For instance, we could improve our results by adding rules that allow us to reason about specific C library functions. For example, the rule:

$$\begin{array}{l} x = atoi(p); \\ y = atoi(q); \end{array} \quad \Longrightarrow \quad \begin{array}{l} y = atoi(q); \\ x = atoi(p); \end{array}$$

can be added because `atoi` does not modify memory. Another example is:

$$\begin{array}{c} memset(p, x, l_1); \\ y = load(getelemptr(p, l_2)) \end{array} \quad \underset{l_2 < l_1}{\Longrightarrow} \quad y = x$$

which enables more aggressive constant propagation. Both of these rules seem to be used by the LLVM optimizer, but we have not added them to our validator at this time. However, adding these sorts of rules is fairly easy, and in a realistic setting many such rules would likely be desired.

Figure 7 shows similar results for loop-invariant code motion (LICM). The baseline LICM, with no rewrite rules, is approximately 75-80%. If we add in all of our rewrite rules, we only improve very slightly. In theory, we should be able to completely validate LICM with no rules. However, again, LLVM uses specific knowledge of certain C library functions. For example, in the following loop:

```
for (int i = 0; i < strlen(p); i++) f(p[i]);
```

the call to `strlen` is known (by LLVM) to be constant. Therefore, LLVM will lift the call to `strlen` out of the loop:

```
int tmp = strlen(p);
for (int i = 0; i < tmp; i++) f(p[i]);
```

Our tool does not have any rules for specific functions, and therefore we do not validate this transformation. The reason why we sometimes get a small improvement in LICM is because very occasionally a rewriting like the one above corresponds to one of our general rules.

Finally, figure 8 shows the effect of rewrite rules on sparse-conditional constant propagation. For this optimization, we used four configurations:

1. no rules

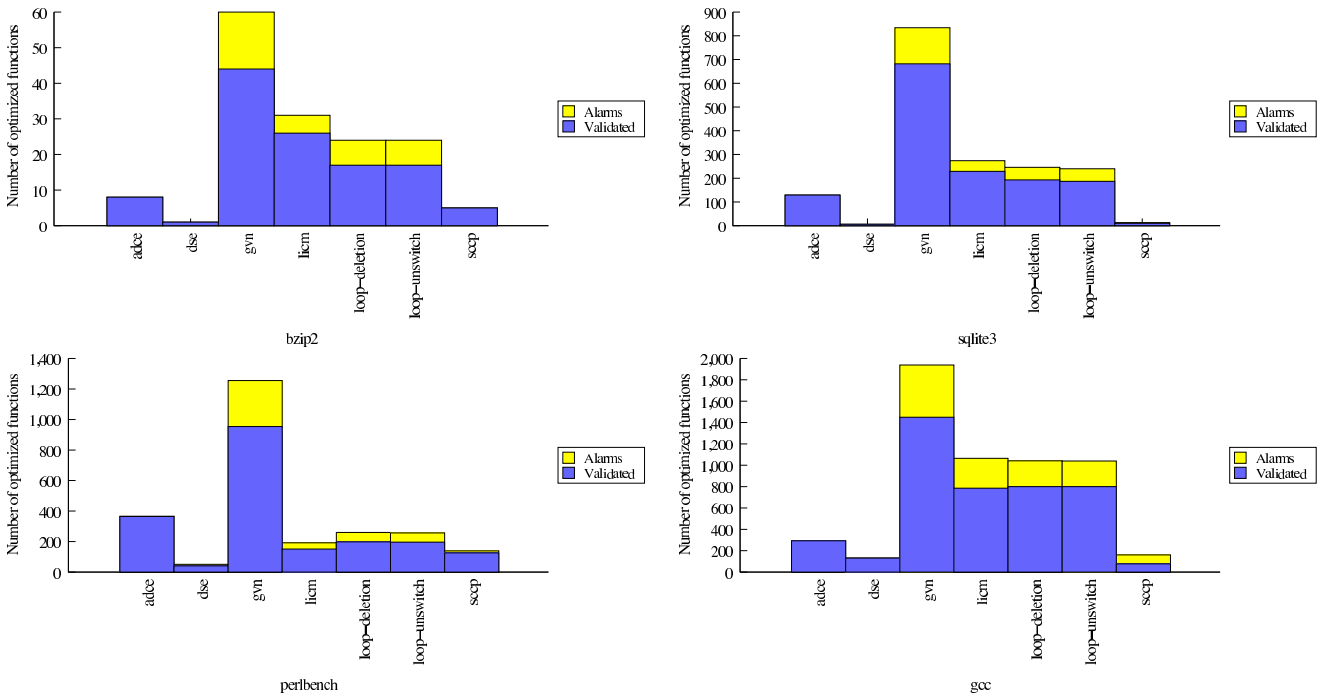


Figure 5: Validator results for individual optimizations

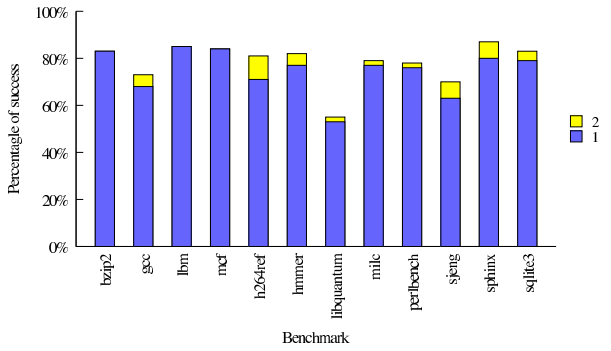


Figure 7: LICM

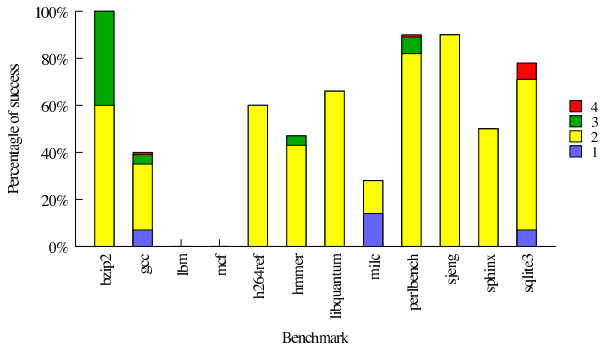


Figure 8: SCCP

2. constant folding
3. ϕ simplification
4. all rules

As expected, with no rules the results are very poor. However, if we add rules for constant folding, we see an immediate improvement, as expected. If we also add rules for reducing ϕ-nodes, bzip2 immediately goes to 100%, even though these rules ave no effect on SQLite. However, additional rules do improve SQLite, but not the other benchmarks.

5.4 Discussion

While implementing our prototype, we were surprised to find that essentially all of the technical difficulties lie in the complex ϕ-nodes. In an earlier version of this work we focused on structured code, and the (binary) ϕ-nodes did not present any real difficulties. However, once we moved away from the structured-code restriction we encountered more problems. First, although the algorithms are known, computing the gates for arbitrary control flow is a fairly involved task. Also, since the gates are dependent on the paths in the CFG, and general C code does not have a simple structured control flow, optimizations will often change the gating conditions even if the control flow is not changed.

Another important aspect of the implementation is the technique for maximizing sharing within the graph. The rewrite rules do a good job of exposing equivalent leaf nodes in the graphs. However, in order to achieve good results, it is important to find equivalent cycles in the graphs and merge them. Again, matching complex ϕ-nodes seems to be the difficult part. To match cycles, we find pairs of μ-nodes in the graph, and trace along their paths in parallel trying to build up a unifying substitution for the graph nodes involved. For ϕ-nodes we sort the branches and conditions and perform a syntactic equality check. This technique is very simple, and

efficient because it only needs to query and update a small portion of the graph.

We also experimented with a Hopcroft partitioning algorithm[7]. Rather than a simple syntactic matching, our partitioning algorithm uses a prolog-style backtracking unification algorithm to find congruences between ϕ-nodes. Surprisingly, the partitioning algorithm with backtracking does not perform better than the simple unification algorithm. Both algorithms give us roughly the same percentage of validation. Our implementation uses the simple algorithm by default, and when this fails it falls back to the slower, partitioning algorithm. Interestingly, this strategy performs slightly better than either technique alone, but not significantly better.

Matching expressions with complex ϕ-nodes seems well within the reach of any SMT prover. Our preliminary experiments with Z3 suggest that it can easily handle the sort of equivalences we need to show. However, this seems like a very heavy-weight tool. One question in our minds is whether or not there is an effective technique somewhere in the middle: more sophisticated than syntactic matching, but short of a full SMT prover.

6. Related work

How do those results compare with the work of Necula and Tate *et al*.? The validator of Necula validates part of GCC 2.7 and the experiments show the results of compiling, and validating, GCC 2.91. It is important to note that the experiments were run on a Pentium Pro machine running at 400 MHz. Four optimizations were considered. Common-subexpression elimination, with a rate of false alarms of roughly 5% and roughly 7 minutes running time. Loop unrolling with a rate of false alarms of 6.3% and roughly 17 minutes running time. Register allocation with a rate of false alarms of 0.1% and around 10 minutes running time. Finally, instruction scheduling with a rate of false alarms of 0.01% and around 9 minutes running time. Unfortunately, the only optimization that we can compare to is CSE as we do not handle loop unrolling, and register allocation is part of the LLVM backend. In theory, we could handle scheduling (with as good results) but LLVM does not have this optimization. For CSE, our results are not as good as Necula's. However, we are dealing with a more complex optimization: global value numbering with partial redundancy elimination and alias information, libc knowledge, and some constant folding. The running times are also similar, but on different hardware. It is therefore unclear whether our validator does better or worse.

The validator of Tate *et al*. validates the Soot research compiler which compiles Java code to the JVM. On SpecJVM they report an impressive rate of alarms of only 2%. However, the version of the Soot optimizer they validate uses more basic optimizations than LLVM, and does not include, for instance, GVN. Given that our results are mostly directed by GVN with alias analysis, it makes comparisons difficult. Moreover, they do not explain whether the number they report takes into account all the functions or only the ones that were actually optimized.

The validator of Kanade *et al*., even though heavily instrumented, is also interesting. They validate GCC 4.1.0 and report no false alarms for CSE, LICM, and copy propagation. To our knowledge, this experiment has the best results. However, it is unclear whether their approach can scale. The authors say that their approach is limited to functions with several hundred RTL instructions and a few hundred transformations. As explained in the introduction, functions with more than a thousand instructions are common in our setting.

There is a wide array of choices for value-graph representations of programs. Weise *et al*. [19] have a nice summary of the various value-graphs. Ours is close to the representation that results from the hash-consed symbolic analysis of a gated SSA graph [6, 18].

7. Conclusion

In conclusion, we believe that normalizing value-graph translation validation of industrial-strengh compilers without instrumentation is feasible. The design relies on well established algorithms and is simple enough to implement. We have been able, in roughly 3 man-months, to build a tool that can validate the optimizations of a decent LLVM pipeline on challenging benchmarks, with a reasonable rate of false alarms. Better yet, we know that many of the false alarms that we witness now require the addition of normalization rules but no significant changes in the design. For instance, insider knowledge of libc functions, floating-points constant folding and folding of global variables are sources of false alarms that can be dealt with by adding normalization rules. There is also room for improvement of the runtime performance of the tool.

There are still a few difficult challenges ahead of us, the most important of which is inter-procedural optimizations. With LLVM, even -O1 makes use of such optimizations and, even though it is clear that simulation-based translation validation can handle inter-procedural optimizations [12], we do not yet know how to precisely generalize normalizing translation. We remark that, in the case of safety-critical code that respects standard code practices [3], as can be produced by tools like Simulink [15], the absence of recursive functions allows us to inline every function (which is reasonable with hash-consing). Preliminary experiments indicate that we are able to validate very effectively inter-procedural optimizations in such a restricted case. Advanced loop transformations are also important, and we believe that this problem may not be as hard as it may seem at first. Previous work [9, 17] has shown that it can be surprisingly easy to validate advanced loop optimizations such as software pipelining with modulo variable expansion if we reason at the value-graph level.

Acknowledgments

We would like to thank Vikram Adve for his early interest and enthusiasm, and Sorin Lerner for discussing this project and exchanging ideas.

References

[1] LLVM 2.8. http://llvm.org.

[2] SQLite 3. http://www.sqlite.org.

[3] The Motor Industry Software Reliability Association. Guidelines for the use of the c language in critical systems. http://www.misra.org.uk, 2004.

[4] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.

[5] SPEC CPU2006. http://www.spec.org/cpu2006/.

[6] Paul Havlak. Construction of thinned gated single-assignment form. In *Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.

[7] John Hopcroft. An n log n algorithm for minimizing states of a finite automaton. In *The Theory of Machines and Computations*, 1971.

[8] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *4th Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.

[9] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[10] E. Moggi. Computational lambda-calculus and monads. In *4th Logic in computer science*, pages 14–23. IEEE, 1989.

[11] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[12] Amir Pnueli and Anna Zaks. Validation of interprocedural optimization. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[13] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[14] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

[15] Simulink. http://mathworks.com.

[16] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *36th Principles of Programming Languages*, pages 264–276. ACM, 2009.

[17] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *37th Principles of Programming Languages*, pages 83–92. ACM Press, 2010.

[18] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Programming Language Design and Implementation*, pages 47–55, 1995.

[19] Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *21st Principles of Programming Languages*, pages 297–310, 1994.