Credible Compilation *

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper presents an approach to compiler correctness in which the compiler generates a proof that the transformed program correctly implements the input program. A simple proof checker can then verify that the program was compiled correctly. We call a compiler that produces such proofs a *credible compiler*, because it produces verifiable evidence that it is operating correctly.

1 Introduction

Today, compilers are black boxes. The programmer gives the compiler a program, and the compiler spits out a bunch of bits. Until he or she runs the program, the programmer has no idea if the compiler has compiled the program correctly. Even running the program offers no guarantees — compiler errors may show up only for certain inputs. So the programmer must simply trust the compiler.

We propose a fundamental shift in the relationship between the compiler and the programmer. Every time the compiler transforms the program, it generates a proof that the transformed program produces the same result as the original program. When the compiler finishes, the programmer can use a simple proof checker to verify that the program was compiled correctly. We call a compiler that generates these proofs a *credible* compiler, because it produces verifiable evidence that it is operating correctly.

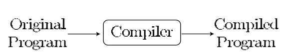

Figure 1: Traditional Compilation

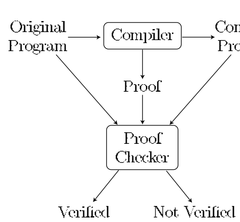

Figure 2: Credible Compilation

*This is an abridged version of the technical report MIT-LCS-TR-776 with the same title, available at www.cag.lcs.mit.edu/~rinard/techreport/credibleCompilation.ps.

Figures 1 and 2 graphically illustrate the difference between traditional compilation and credible compilation. A traditional compiler generates a compiled program and nothing else. A credible compiler, on the other hand, also generates a proof that the compiled program correctly implements the original program. A proof checker can then take the original program, the proof, and the compiled program, and check if the proof is correct. If so, the compilation is verified and the compiled program is guaranteed to correctly implement the original program. If the proof does not check, the compilation is not verified and all bets are off.

2 Example

In this section we present an example that explains how a credible compiler can prove that it performed a translation correctly. Figure 3 presents the example program represented as a control flow graph. The program contains several assignment nodes; for example the node $5 : i \leftarrow i + x + y$ at label 5 assigns the value of the expression $i+x+y$ to the variable i. There is also a conditional branch node $4 : \text{br } i < 24$. Control flows from this node through its outgoing left edge to the assignment node at label 5 if $i < 24$, otherwise control flows through the right edge to the exit node at label 7.

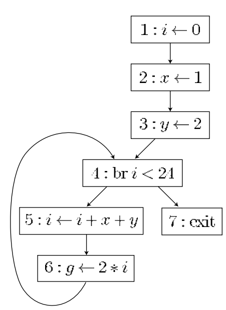

Figure 3: Original Program

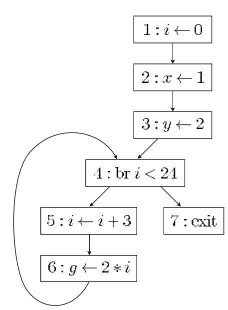

Figure 4: Program After Constant Propagation and Constant Folding

Figure 4 presents the program after constant propagation and constant folding. The compiler has replaced the node $5 : i \leftarrow i + x + y$ at label 5 with the node $5 : i \leftarrow i + 3$. The goal is to prove that this particular transformation on this particular program preserves the semantics of the original program. The goal is *not* to prove that the compiler will always transform an arbitrary program correctly.

To perform this optimization, the compiler did two things:

- **Analysis:** The compiler determined that x is always 1 and y is always 2 at the program point before node 5. So, $x + y$ is always 3 at this program point.

- **Transformation:** The compiler used the analysis information to transform the program so that generates the same result while (hopefully) executing in less time or space or consuming less power. In our example, the compiler simplifies the expression $x + y$ to 3.

Our approach to proving optimizations correct supports this basic two-step structure. The compiler first proves that the analysis is correct, then uses the analysis results to prove that the original and transformed programs generate the same result. Here is how this approach works in our example.

2.1 Proving Analysis Results Correct

Many years ago, Floyd came up with a technique for proving properties of programs [4]. This technique was generalized and extended, and eventually came to be understood as a logic whose proof rules are derived from the structure of the program [2]. The basic idea is to assert a set of properties about the relationships between variables at different points in the program, then use the logic to prove that the properties always hold. If so, each property is called an invariant, because it is always true when the flow of control reaches the corresponding point in the program.

In our example, the key invariant is that at the point just before the program executes node 5, it is always true that $x = 1$ and $y = 2$. We represent this invariant as $\langle x = 1 \wedge y = 2\rangle 5$.

In our example, the simplest way for the compiler to generate a proof of $\langle x = 1 \wedge y = 2\rangle 5$ is for it to generate a set of invariants that represent the analysis results, then use the logic to prove that all of the invariants hold. Here is the set of invariants in our example:

- $\langle x = 1\rangle 3$
- $\langle x = 1 \wedge y = 2\rangle 4$
- $\langle x = 1 \wedge y = 2\rangle 5$
- $\langle x = 1 \wedge y = 2\rangle 6$

Conceptually, the compiler proves this set of invariants by tracing execution paths. The proof is by induction on the structure of the partial executions of the program. For each invariant, the compiler first assumes that the invariants at all preceding nodes in the control flow graph are true. It then traces the execution through each preceding node to verify the invariant at the next node. We next present an outline of the proofs for several key invariants.

- $\langle x = 1\rangle 3$ because the only preceding node, node 2, sets x to 1.

- To prove $\langle x = 1 \wedge y = 2\rangle 4$, first assume $\langle x = 1\rangle 3$ and $\langle x = 1 \wedge y = 2\rangle 6$. Then consider the two preceding nodes, nodes 3 and 6. Because $\langle x = 1\rangle 3$ and 3 sets y to 2, $\langle x = 1 \wedge y = 2\rangle 4$. Because $\langle x = 1 \wedge y = 2\rangle 6$ and node 6 does not affect the value of either x or y, $\langle x = 1 \wedge y = 2\rangle 4$.

In this proof we have assumed that the compiler generates an invariant at almost all of the nodes in the program. More traditional approaches use fewer invariants, typically one invariant per loop, then produce proofs that trace paths consisting of multiple nodes.

2.2 Proving Transformations Correct

When a compiler transforms a program, there are typically some externally observable effects that it must preserve. A standard requirement, for example, is that the compiler must preserve the input/output relation of the program. In our framework, we assume that the compiler is operating on a compilation unit such as procedure or method, and that there are externally observable variables such as global variables or object instance variables. The compiler must preserve the final values of these variables. All other variables are either parameters or local variables, and the compiler is free to do whatever it wants to with these variables so long as it preserves the final values of the observable variables. The compiler may also assume that the initial values of the observable variables and the parameters are the same in both cases.

In our example, the only requirement is that the transformation must preserve the final value of the variable g. The compiler proves this property by proving a simulation correspondence between the original and transformed programs. To present the correspondence, we must be able to refer, in the same context, to variables and node labels from the two programs. We adopt the convention that all entities from the original program P will have a subscript of P, while all entities from the transformed program T will have a subscript of T. So i_P refers to the variable i in the original program, while i_T refers to the variable i in the transformed program.

In our example, the compiler proves that the transformed program simulates the original program in the following sense: for every execution of the original program P that reaches the final node 7_P, there exists an execution of the transformed program T that reaches the final node 7_T such that g_P at $7_P = g_T$ at 7_T. We call such a correspondence a *simulation invariant*, and write it as $\langle g_P\rangle 7_P \triangleright \langle g_T\rangle 7_T$.

The compiler typically generates a set of simulation invariants, then uses the logic to construct a proof of the correctness of all of the simulation invariants. The proof is by induction on the length of the partial executions of the original program. We next outline how the compiler can use this approach to prove $\langle g_P\rangle 7_P \triangleright \langle g_T\rangle 7_T$. First, the compiler is given that $\langle g_P\rangle 1_P \triangleright \langle g_T\rangle 1_T$ — in other words, the values of g_P and g_T are the same at the start of the two programs. The compiler then generates the following simulation invariants:

- $\langle (g_P, i_P)\rangle 2_P \triangleright \langle (g_T, i_T)\rangle 2_T$
- $\langle (g_P, i_P)\rangle 3_P \triangleright \langle (g_T, i_T)\rangle 3_T$
- $\langle (g_P, i_P)\rangle 4_P \triangleright \langle (g_T, i_T)\rangle 4_T$
- $\langle (g_P, i_P)\rangle 5_P \triangleright \langle (g_T, i_T)\rangle 5_T$
- $\langle (g_P, i_P)\rangle 6_P \triangleright \langle (g_T, i_T)\rangle 6_T$
- $\langle g_P\rangle 7_P \triangleright \langle g_T\rangle 7_T$

The key simulation invariants are $\langle g_P\rangle 7_P \triangleright \langle g_T\rangle 7_T$, $\langle (g_P, i_P)\rangle 6_P \triangleright \langle (g_T, i_T)\rangle 6_T$ and $\langle (g_P, i_P)\rangle 4_P \triangleright \langle (g_T, i_T)\rangle 4_T$. We next outline the proofs of these two invariants.

- To prove $\langle g_P\rangle 7_P \triangleright \langle g_T\rangle 7_T$, first assume $\langle (g_P, i_P)\rangle 4_P \triangleright \langle (g_T, i_T)\rangle 4_T$. For each path to 7_P in P, we must find a corresponding path in T to 7_P such that the values of g_P and g_T are the same in both paths. The only path to 7_P goes from 4_P to 7_P when $i_P \geq 24$. The corresponding path in T goes from 4_T to 7_T when $i_T \geq$

24. Because $\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T$, control flows from 4_T to 7_T whenever control flows from 4_P to 7_P. The simulation invariant $\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T$ also implies that the values of g_P and g_T are the same in both cases.

- To prove $\langle(g_P, i_P)\rangle 6_P \rhd \langle(g_T, i_T)\rangle 6_T$, assume $\langle(g_P, i_P)\rangle 5_P \rhd \langle(g_T, i_T)\rangle 5_T$. The only path to 6_P goes from 5_P to 6_P, with i_P at $6_P = i_P$ at $5_P + x_P$ at $5_P + y_P$ at 5_P. The analysis proofs showed that x_P at $5_P + y_P$ at $5_P = 3$, so i_P at $6_P = i_P$ at $5_P + 3$. The corresponding path in T goes from 5_T to 6_T, with i_T at $6_T = i_T$ at $5_T + 3$.

  The assumed simulation invariant $\langle(g_P, i_P)\rangle 5_P \rhd \langle(g_T, i_T)\rangle 5_T$ allows us verify a correspondence between the values of i_P at 6_P and i_T at 6_P; namely that they are equal. Because 5_P does not change g_P and 5_T does not change g_T, g_P at 6_P and g_T at 6_P have the same value.

- To prove $\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T$, first assume $\langle(g_P, i_P)\rangle 3_P \rhd \langle(g_T, i_T)\rangle 3_T$ and $\langle(g_P, i_P)\rangle 6_P \rhd \langle(g_T, i_T)\rangle 6_T$. There are two paths to 4_P:

  - Control flows from 3_P to 4_P. The corresponding path in T is from 3_T to 4_T, so we can apply the assumed simulation invariant $\langle(g_P, i_P)\rangle 3_P \rhd \langle(g_T, i_T)\rangle 3_T$ to derive g_P at $4_P = g_T$ at 4_T and i_P at $4_P = i_T$ at 4_T.
  - Control flows from 6_P to 4_P, with g_P at $4_P = 2 * i_P$ at 6_P. The corresponding path in T is from 6_T to 4_T, with g_T at $4_T = 2 * i_T$ at 6_T. We can apply the assumed simulation invariant $\langle(g_P, i_P)\rangle 6_P \rhd \langle(g_T, i_T)\rangle 6_T$ to derive $2 * i_P$ at $6_P = 2 * i_T$ at 6_T. Since 6_P does not change i_P and 6_T does not change i_T, we can derive g_P at $4_P = g_T$ at 4_T and i_P at $4_P = i_T$ at 4_T.

2.3 Formal Foundations

In this section we outline the formal foundations required to support credible compilation. The full paper presents the formal foundations in detail [11].

Credible compilation depends on machine-checkable proofs. Its use therefore requires the formalization of several components. First, we must formalize the program representation, namely control-flow graphs, and define a formal operational semantics for that representation. To support proofs of standard invariants, we must formalize the Floyd-Hoare proof rules required to prove any program property. We must also formalize the proof rules for simulation invariants. Both of these formalizations depend on some facility for doing simple automated proofs of standard facts involving integer properties. It is also necessary to show that both sets of proof rules are sound.

We have successfully performed these tasks for a simple language based on control-flow graphs; see the full report for the complete elaboration [11]. Given this formal foundation, the compiler can use the proof rules to produce machine-checkable proofs of the correctness of its analyses and transformations.

3 Optimization Schemas

We next present examples that illustrate how to prove the correctness of a variety of standard optimizations. Our goal is to establish a general schema for each optimization. The compiler would then use the schema to produce a correctness proof that goes along with each optimization.

3.1 Dead Assignment Elimination

The compiler can eliminate an assignment to a local variable if that variable is not used after the assignment. The proof schema is relatively simple: the compiler simply generates simluation invariants that assert the equality of corresponding live variables at corresponding points in the program. Figures 5 and 6 present an example that we use to illustrate the schema. This example continues the example introduced in Section 2. Figure 7 presents the invariants that the compiler generates for this example.

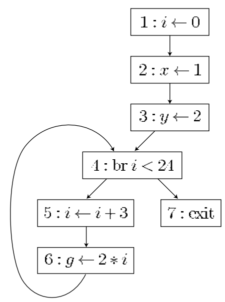

Figure 5: Program P Before Dead Assignment Elimination

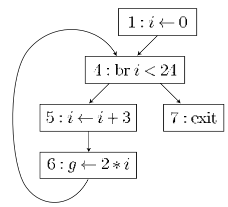

Figure 6: Program T After Dead Assignment Elimination

$$
\begin{aligned}
I \;=\; & \{\langle(g_P, i_P)\rangle 4_P \rhd \langle(g_T, i_T)\rangle 4_T, \langle i_P\rangle 5_P \rhd \langle i_T\rangle 5_T, \\
& \langle i_P\rangle 6_P \rhd \langle i_T\rangle 6_T, \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T\}
\end{aligned}
$$

Figure 7: Invariants for Dead Assignment Elimination

Note that the set I of invariants contains no standard invariants. In general, dead assignment elimination requires only simulation invariants. The proofs of these invariants are simple; the only complication is the need to skip over dead assignments.

3.2 Branch Movement

Our next optimization moves a conditional branch from the top of a loop to the bottom. The optimization is legal if the loop always executes at least once. This optimization is different from all the other optimizations we have discussed so far in that it changes the control flow. Figure 8 presents the program before branch movement; Figure 9 presents the program after branch movement. Figure 10 presents the set of invariants that the compiler generates for this example.

One of the paths that the proof must consider is the path in the original program P from 1_P to 4_P to 7_P. No execution of P, of course, will take this path — the loop always executes at least once, and this path corresponds to the loop executing zero times. The fact that this path will never execute shows up as a false condition in the partial simulation invariant for P that is propagated from 7_P back to 1_P. The corresponding path in T that is used to prove $I \vdash \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T$ is the path from 1_T through 5_T, 6_T, and 4_T to 7_T.

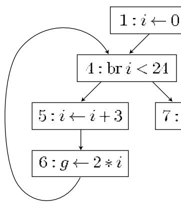

Figure 8: Program P Before Branch Movement

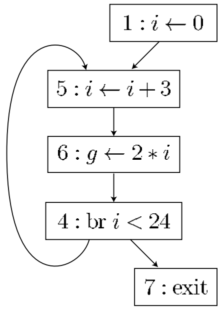

Figure 9: Program T After Branch Movement

$$I = \{\langle i_P\rangle 5_P \rhd \langle i_T\rangle 5_T, \langle i_P\rangle 6_P \rhd \langle i_T\rangle 6_T, \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T\}$$

Figure 10: Invariants for Branch Movement

3.3 Induction Variable Elimination

Our next optimization eliminates the induction variable i from the loop, replacing it with g. The correctness of this transformation depends on the invariant $\langle g_P = 2 * i_P\rangle 4_P$. Figure 11 presents the program before induction variable elimination; Figure 12 presents the program after induction variable elimination. Figure 13 presents the set of invariants that the compiler generates for this example. These invariants characterize the relationship between the eliminated induction variable i_P from the original program and the variable g_T in the transformed program.

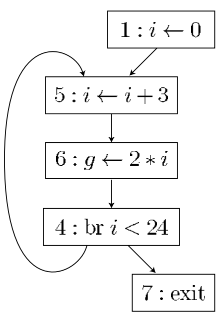

Figure 11: Program P Before Induction Variable Elimination

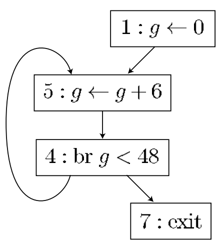

Figure 12: Program T After Induction Variable Elimination

$$\begin{aligned} I \;=\; & \{\langle g_P = 2 * i_P\rangle 4_P, \langle 2 * i_P\rangle 5_P \rhd \langle g_T\rangle 5_T, \\ & \langle 2 * i_P\rangle 4_P \rhd \langle g_T\rangle 4_T, \langle g_P\rangle 7_P \rhd \langle g_T\rangle 7_T\} \end{aligned}$$

Figure 13: Invariants for Induction Variable Elimination

3.4 Loop Unrolling

The next optimization unrolls the loop once. Figure 14 presents the program before loop unrolling; Figure 15 presents the program after unrolling the loop. Note that the loop unrolling transformation preserves the loop exit test; this test can be eliminated by the dead code elimination optimization discussed in Section 3.5.

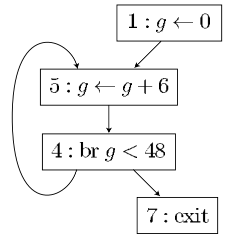

Figure 14: Program P Before Loop Unrolling

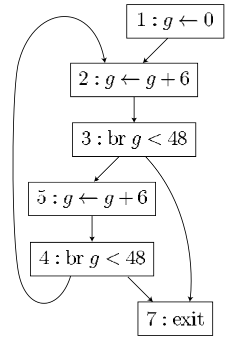

Figure 15: Program T After Loop Unrolling

$$\begin{aligned} I \;=\; & \{\langle g_P\%12 = 0 \vee g_P\%12 = 6\rangle 4_P, \langle g_P\%12 = 0, g_P\rangle 5_P \\ & \quad \rhd \langle g_T\rangle 2_T, \langle g_P\%12 = 6, g_P\rangle 4_P \rhd \langle g_T\rangle 3_T, \\ & \langle g_P\%12 = 6, g_P\rangle 5_P \rhd \langle g_T\rangle 5_T, \\ & \langle g_P\%12 = 0, g_P\rangle 4_P \rhd \langle g_T\rangle 4_T, \langle g_P\rangle 7_P \rhd \langle g_P\rangle 7_P\} \end{aligned}$$

Figure 16: Invariants for Loop Unrolling

Figure 16 presents the set of invariants that the compiler generates for this example. Note that, unlike the simulation invariants in previous examples, these simulation invariants have conditions. The conditions are used to separate different executions of the same node in the original program. Some of the time, the execution at node 4_P corresponds to the execution at node 4_T, and other times to the execution at node 3_T. The conditions in the simulation invariants identify when, in the execution of the original program, each correspondence holds. For example, when $g_P\%12 = 0$, the execution at 4_P corresponds to the execution at 4_T; when $g_P\%12 = 6$, the execution at 4_P corresponds to the execution at 3_T.

3.5 Dead Code Elimination

Our next optimization is dead code elimination. We continue with our example by eliminating the branch in the middle of the loop at node 3. Figure 17 presents the program before the branch is eliminated. The key property that allows the compiler to remove the branch is that $g\%12 = 6 \wedge g \leq 48$ at 3, which implies that $g < 48$ at 3. In other words, the condition in the branch is always true. Figure 18 presents the program after the branch is eliminated. Figure 19 presents the set of invariants that the compiler generates for this example.

One of the paths that the proof must consider is the potential loop exit in the original program P from 3_P to 7_P; In fact, the loop always exits from 4_P, not 3_P. This fact shows up because the conjunction of the standard invariant $\langle g_P\%12 = 6 \wedge g_P \leq 48\rangle 3_P$ with the condition $g_P \geq 48$ from the partial simulation invariant for P at 3_P is false. The corresponding path in T that is used to prove $I \vdash \langle i_P\rangle 7_P \rhd \langle i_T\rangle 7_T$ is the path from 5_T to 4_T to 7_T.

4 Code Generation

In principle, we believe that it is possible to produce a proof that the final object code correctly implements the original program. For engineering reasons, however, we designed the

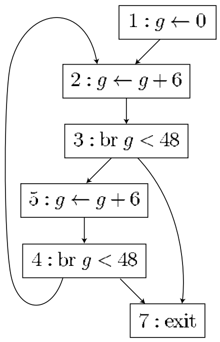

Figure 17: Program P Before Dead Code Elimination

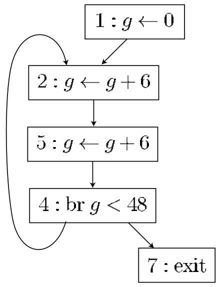

Figure 18: Program T After Dead Code Elimination

$$\begin{aligned} I \;=\; & \{\langle g_P\%12 = 0 \wedge g_P < 48\rangle 2_P, \\ & \langle g_P\%12 = 6 \wedge g_P \leq 48\rangle 3_P, \\ & \langle g_P\%12 = 6 \wedge g_P < 48\rangle 5_P, \\ & \langle g_P\%12 = 0 \wedge g_P \leq 48\rangle 4_P, \langle g_P\rangle 2_P \rhd \langle g_P\rangle 2_P, \\ & \langle g_P\rangle 5_P \rhd \langle g_P\rangle 5_P, \\ & \langle g_P\rangle 3_P \rhd \langle g_P\rangle 5_P, \langle g_P\rangle 4_P \rhd \langle g_P\rangle 4_P, \\ & \langle g_P\rangle 7_P \rhd \langle g_P\rangle 7_P\} \end{aligned}$$

Figure 19: Invariants for Dead Code Elimination

proof system to work with a standard intermediate format based on control flow graphs. The parser, which produces the initial control flow graph, and the code generator, which generates object code from the final control flow graph, are therefore potential sources of uncaught errors. We believe it should be straightforward, for reasonable languages, to produce a standard parser that is not a serious source of errors. It is not so obvious how the code generator can be made simple enough to be reliable.

Our goal is make the step from the final control flow graph to the generated code be as small as possible. Ideally, each node in the control flow graph would correspond to a single instruction in the generated code. To achieve this goal, it must be possible to express the result of complicated, machine-specific code generation algorithms (such as register allocation and instruction selection) using control flow graphs. After the compiler applies these algorithms, the final control flow graph would be structured in a stylized way appropriate for the target architecture. The code generator for the target architecture would accept such a control flow graph as input and use a simple translation algorithm to produce the final object code.

With this approach, we anticipate that code generators can be made approximately as simple as proof checkers. We therefore anticipate that it will be possible to build standard code generators with an acceptable level of reliability for most users. However, we would once again like to emphasize that it should be possible to build a framework in which the compilation is checked from source code to object code.

In the following two sections, we first present an approach for a simple RISC instruction set, then discuss an approach for more complicated instruction sets.

4.1 A Simple RISC Instruction Set

For a simple RISC instruction set, the key idea is to introduce special variables that the code generator interprets as registers. The control flow graph is then transformed so that each node corresponds to a single instruction in the generated code. We first consider assignment nodes.

- If the destination variable is a register variable, the source expression must be one of the following:
  - A non-register variable. In this case the node corresponds to a load instruction.
  - A constant. In this case the node corresponds to a load immediate instruction.
  - A single arithmetic operation with register variable operands. In this case the node corresponds to an arithmetic instruction that operates on the two source registers to produce a value that is written into the destination register.
  - A single arithmetic operation with one register variable operand and one constant operand. In this case the node corresponds to an arithmetic instruction that operates on one source register and an immediate constant to produce a value that is written into the destination register.
- If the destination variable of an assignment node is a non-register variable, the source expression must consist of a register variable, and the node corresponds to a store instruction.

It is possible to convert assignment nodes with arbitrary expressions to this form. The first step is to flatten the expression by introducing temporary variables to hold the intermediate values computed by the expression. Additional assignment nodes transfer these values to the new temporary variables. The second step is to use a register allocation algorithm to transform the control flow graph to fit the form described above.

We next consider conditional branch nodes. If the condition is the constant true or false, the node corresponds to an unconditional branch instruction. Otherwise, the condition must compare a register variable with zero so that the instruction corresponds either to a branch if zero instruction or a branch if not zero instruction.

4.2 More Complex Instruction Sets

Many processors offer more complex instructions that, in effect, do multiple things in a single cycle. In the ARM instruction set, for example, the execution of each instruction may be predicated on several condition codes. ARM instructions can therefore be modeled as consisting of a conditional branch plus the other operations in the instruction. The x86 instruction set has instructions that assign values to several registers.

We believe the correct approach for these more complex instruction sets is to let the compiler writer extend the possible types of nodes in the control flow graph. The semantics of each new type of node would be given in terms of the base nodes in standard control flow graphs. We illustrate this approach with an example.

For instruction sets with condition codes, the programmer would define a new variable for each condition code and new assignment nodes that set the condition codes appropriately. The semantics of each new node would be given as a small control flow graph that performed the assignment, tested the appropriate conditions, and set the appropriate condition code variables. If the instruction set also has predicated execution, the control flow graph would use conditional branch nodes to check the appropriate condition codes before performing the instruction.

Each new type of node would come with proof rules automatically derived from its underlying control flow graph. The proof checker could therefore verify proofs on control flow graphs that include these types of nodes. The code generator would require the preceding phases of the compiler to produce a control flow graph that contained only those types of nodes that translate directly into a single instruction on the target architecture. With this approach, all complex code generation algorithms could operate on control flow graphs, with their results checked for correctness.

5 Related Work

Most existing research on compiler correctness has focused on techniques that deliver a compiler guaranteed to operate correctly on every input program [5]; we call such a compiler a *totally correct* compiler. A credible compiler, on the other hand, is not necessarily guaranteed to operate correctly on all programs — it merely produces a proof that it has operated correctly on the current program.

In the absence of other differences, one would clearly prefer a totally correct compiler to a credible compiler. After all, the credible compiler may fail to compile some programs correctly, while the totally correct compiler will always work. But the totally correct compiler approach imposes a significant pragmatic drawback: it requires the source code of the compiler, rather than its output, to be proved correct. So programmers must express the compiler in a way that is amenable to these correctness proofs. In practice this invasive constraint has restricted the compiler to a limited set of source languages and compiler algorithms. Although the concept of a totally correct compiler has been around for many years, there are, to our knowledge, no totally correct compilers that produce close to production-quality code for realistic programming languages. Credible compilation offers the compiler developer much more freedom. The compiler can be developed in any language using any methodology and perform arbitrary transformations. The only constraint is that the compiler produce a proof that its result is correct.

The concept of credible compilers has also arisen in the context of compiling synchronous languages [3, 8]. Our approach, while philosophically similar, is technically much different. It is designed for standard imperative languages and therefore uses drastically different techniques for deriving and expressing the correctness proofs.

We often are asked the question "How is your approach different from proof-carrying code [6]?"* In our view, credible compilers and proof-carrying code are orthogonal concepts. Proof-carrying code is used to prove properties of *one* program, typically the compiled program. Credible compilers establish a correspondence between *two* programs: an original program and a compiled program. Given a safe programming language, a credible compiler will produce guarantees that are stronger than those provided by typical applications of proof-carrying code. So, for example, if the source language is type safe and a credible compiler produces a proof that the compiled program correctly implements the original program, then the compiled program is also type safe.

But proof-carrying code can, in principle, be used to prove properties that are not visible in the semantics of the language. For example, one might use proof-carrying code to prove that a program does not execute a sequence of instructions that may damage the hardware. Because most languages simply do not deal with the kinds of concepts required to prove such a property as a correspondence between two programs, credible compilation is not particularly relevant to these kinds of problems.

Since we first wrote this paper, credible compilation has developed significantly. For example, it has been applied to programs with pointers [10] and to C programs [7].

*Proof-carrying code is code augmented with a proof that the code satisfies safety properties such as type safety or the absence of array bounds violations.

6 Impact of Credible Compilation

We next discuss the potential impact of credible compilation. We consider five areas: debugging compilers, increasing the flexibility of compiler development, just-in-time compilers, concept of an open compiler, and the relationship of credible compilation to building custom compilers.

6.1 Debugging Compilers

Compilers are notoriously difficult to build and debug. In a large compiler, a surprising part of the difficulty is simply recognizing incorrectly generated code. The current state of the art is to generate code after a set of passes, then test that the generated code produces the same result as the original code. Once a piece of incorrect code is found, the developer must spend time tracing the bug back through layers of passes to the original source.

Requiring the compiler to generate a proof for each transformation will dramatically simplify this process. As soon as a pass operates incorrectly, the developer will immediately be directed to the incorrect code. Bugs can be found and eliminated as soon as they occur.

6.2 Flexible Compiler Development

It is difficult, if not impossible, to eliminate all of the bugs in a large software system such as a compiler. Over time, the system tends to stabilize around a relatively reliable software base as it is incrementally debugged. The price of this stability is that people become extremely reluctant to change the software, either to add features or even to fix relatively minor bugs, for fear of inadvertantly introducing new bugs. At some point the system becomes obsolete because the developers are unable to upgrade it quickly enough for it to stay relevant.

Credible compilation, combined with the standard organization of the compiler as a sequence of passes, promises to make it possible to continually introduce new, unreliable code into a mature compiler without compromising functionality or reliability. Consider the following scenario. Working under deadline pressure, a compiler developer has come up a prototype implementation of a complex transformation. This transformation is of great interest because it dramatically improves the performance of several SPEC benchmarks. But because the developer cut corners to get the implementation out quickly, it is unreliable. With credible compilation, this unreliability is not a problem at all — the transformation is introduced into the production compiler as another pass, with the compiler driver checking the correctness proof and discarding the results if it didn't work. The compiler operates as reliably as it did before the introduction of the new pass, but when the pass works, it generates much better code.

It is well known that the effort required to make a compiler work on all conceivable inputs is much greater than the effort required to make the compiler work on all likely inputs. Credible compilation makes it possible to build the entire compiler as a sequence of passes that work only for common or important cases. Because developers would be under no pressure to make passes work on all cases, each pass could be hacked together quickly with little testing and no complicated code to handle exceptional cases. The result is that the compiler would be much easier and cheaper to build and much easier to target for good performance on specific programs.

A final extrapolation is to build speculative transformations. The idea is that the compiler simply omits the analysis required to determine if the transformation is legal. It does the transformation anyway and generates a proof that the transformation is correct. This proof is valid, of course, only if the transformation is correct. The proof checker filters out invalid transformations and keeps the rest.

This approach shifts work from the developer to the proof checker. The proof checker does the analysis required to determine if the transformation is legal, and the developer can focus on the transformation and the proof generation, not on writing the analysis code.

6.3 Just-In-Time Compilers

The increased network interconnectivity resulting from the deployment of the Internet has enabled and promoted a new way to distribute software. Instead of compiling to native machine code that will run only on one machine, the source program is compiled to a portable byte code. An interpreter executes the byte code.

The problem is that the interpreted byte code runs much slower than native code. The proposed solution is to use a just-in-time compiler to generate native code either when the byte code arrives or dynamically as it runs. Dynamic compilation also has the advantage that it can use dynamically collected profiling information to drive the compilation process.

Note, however, that the just-in-time compiler is another complex, potentially erroneous software component that can affect the correct execution of the program. If a compiler generates native code, the only subsystems that can change the semantics of the native code binary during normal operation are the loader, dynamic linker, operating system and hardware, all of which are relatively static systems. An organization that is shipping software can generate a binary and test it extensively on the kind of systems that its customers will use. If the customer finds an error, the organization can investigate the problem by running the program on a roughly equivalent system.

But with dynamic compilation, the compiled code constantly changes in a way that may be very difficult to reproduce. If the dynamic compiler incorrectly compiles the program, it may be extremely difficult to reproduce the conditions that caused it to fail. This additional complexity in the compilation approach makes it more difficult to build a reliable compiler. It also makes it difficult to assign blame for any failure. When an error shows up, it could be either the compiler or the application. The organizations that built each product tend to blame each other for the error, and neither one is motivated to work hard to find and fix the problem. The end result is that the total system stays broken.

Credible compilation can eliminate this problem. If the dynamic compiler emits a proof that it executed correctly, the run-time system can check the proof before accepting the generated code. All incorrect code would be filtered out before it caused a problem. This approach restores the reliability properties of distributing native code binaries while supporting the convenience and flexibility of dynamic compilation and the distribution of software in portable byte-code format.

6.4 An Open Compiler

We believe that credible compilers will change the social context in which compilers are built. Before a developer can safely integrate a pass into the compiler, there must be some evidence that pass will work. But there is currently no way to verify the correctness of the pass. Developers are therefore typically reduced to relying on the reputation of the person that produced the pass, rather than on the trustworthiness of the code itself. In practice, this means that the entire compiler is typically built by a small, cohesive group of people in a single organization. The compiler is closed in the sense that these people must coordinate any contribution to the compiler.

Credible compilation eliminates the need for developers to trust each other. Anyone can take any pass, integrate into their compiler, and use it. If a pass operates incorrectly, it is immediately apparent, and the compiler can discard the transformation. There is no need to trust anyone. The compiler is now open and anyone can contribute. Instead of relying on a small group of people in one organization, the effort, energy, and intelligence of every compiler developer in the world can be productively applied to the development of one compiler.

The keys to making this vision a reality are a standard intermediate representation, logics for expressing the proofs, and a verifier that checks the proofs. The representation must be expressive and support the range of program representations required for both high level and low level analyses and transformations. Ideally, the representation would be extensible, with developers able to augment the system with new constructs and new axioms that characterize these constructs. The verifier would be a standard piece of software. We expect several independent verifiers to emerge that would be used by most programmers; paranoid programmers can build their own verifier. It might even be possible to do a formal correctness proof of the verifier.

Once this standard infrastructure is in place, we can leverage the Internet to create a compiler development community. One could imagine, for example, a compiler development web portal with code transformation passes, front ends, and verifiers. Anyone can download a transformation; anyone can use any of the transformations without fear of obtaining an incorrect result. Each developer can construct his or her own custom compiler by stringing together a sequence of optimization passes from this web site. One could even imagine an intellectual property market emerging, as developers license passes or charge electronic cash for each use of a pass. In fact, future compilers may consist of a set of transformations distributed across multiple web sites, with the program (and its correctness proofs) flowing through the sites as it is optimized.

6.5 Custom Compilers

Compilers are traditionally thought of and built as general-purpose systems that should be able to compile any program given to them. As a consequence, they tend to contain analyses and transformations that are of general utility and almost always applicable. Any extra components would slow the compiler down and increase the complexity.

The problem with this situation is that general techniques tend to do relatively pedestrian things to the program. For specific classes of programs, more specialized analyses and transformations would make a huge difference [12, 9, 1]. But because they are not generally useful, they don't make it into widely used compilers.

We believe that credible compilation can make it possible to develop lots of different custom compilers that have been specialized for specific classes of applications. The idea is to make a set of credible passes available, then allow the compiler builder to combine them in arbitrary ways. Very specialized passes could be included without threatening the stability of the compiler. One could easily imagine a range of compilers quickly developed for each class of applications.

It would even be possible extrapolate this idea to include optimistic transformations. In some cases, it is difficult to do the analysis required to perform a specific transformation. In this case, the compiler could simply omit the analysis, do the transformation, and generate a proof that would be correct if the analysis would have succeeded. If the transformation is incorrect, it will be filtered out by the compiler driver. Otherwise, the transformation goes through.

This example of optimistic transformations illustrates a somewhat paradoxical property of credible compilation. Even though credible compilation will make it much easier to develop correct compilers, it also makes it practical to release much buggier compilers. In fact, as described below, it may change the reliability expectations for compilers.

Programmers currently expect that the compiler will work correctly for every program that they give it. And you can see that something very close to this level of reliability is required if the compiler fails silently when it fails — it is very difficult for programmers to build a system if there is a reasonable probability that a given error can be caused by the compiler and not the application.

But credible compilation completely changes the situation. If the programmer can determine whether or not the the compiler operated correctly before testing the program, the development process can tolerate a compiler that occasionally fails.

In this scenario, the task of the compiler developer changes completely. He or she is no longer responsible for delivering a program that works almost all of the time. It is enough to deliver a system whose failures do not significantly hamper the development of the system. There is little need to make very uncommon cases work correctly, especially if there are known work-arounds. The result is that compiler developers can be much more aggressive — the length of the develoment cycle will shrink and new techniques will be incorporated into production compilers much more quickly.

7 Conclusions

Most research on compiler correctness has focused on obtaining a compiler that is guaranteed to generate correct code for every input program. This paper presents a less ambitious, but hopefully much more practical approach: require the compiler to generate a proof that the generated code correctly implements the input program. Credible compilation, as we call this approach, gives the compiler developer maximum flexibility, helps developers find compiler bugs, and eliminates the need to trust the developers of compiler passes.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[2] K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1997.

[3] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 202–213, Haifa, Israel, June 1997.

[4] R. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Proceedings of the Symposium in Applied Mathematics*, number 19, pages 19–32, 1967.

[5] J. Guttman, J. Ramsdell, and M. Wand. VLISP: a verified implementation of scheme. *Lisp and Symbolic Computing*, 8(1–2):33–110, March 1995.

[6] G. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[7] G. Necula. Translation validation for an optimizing compiler. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[8] A. Pnueli, M. Siegal, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lisbon, Portugal, March 1998.

[9] M. Rinard and P. Diniz. Commutativity analysis: A new framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 54–67, Philadelphia, PA, May 1996. ACM, New York.

[10] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[11] Martin Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1999.

[12] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.