The Design and Implementation of a Certifying Compiler

George C. Necula Peter Lee

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891
{necula,petel}@cs.cmu.edu

Abstract

This paper presents the design and implementation of a compiler that translates programs written in a type-safe subset of the C programming language into highly optimized DEC Alpha assembly language programs, and a *certifier* that automatically checks the type safety and memory safety of any assembly language program produced by the compiler. The result of the certifier is either a formal proof of type safety or a counterexample pointing to a potential violation of the type system by the target program. The ensemble of the compiler and the certifier is called a *certifying compiler*.

Several advantages of certifying compilation over previous approaches can be claimed. The notion of a certifying compiler is significantly easier to employ than a formal compiler verification, in part because it is generally easier to verify the correctness of the result of a computation than to prove the correctness of the computation itself. Also, the approach can be applied even to highly optimizing compilers, as demonstrated by the fact that our compiler generates target code, for a range of realistic C programs, which is competitive with both the cc and gcc compilers with all optimizations enabled. The certifier also drastically improves the effectiveness of compiler testing because, for each test case, it statically signals compilation errors that might otherwise require many executions to detect. Finally, this approach is a practical way to produce the safety proofs for a Proof-Carrying Code system, and thus may be useful in a system for safe mobile code.

1 Introduction

The question of compiler correctness is as old as the first compiler implementations. In a paper published in 1963, John McCarthy refers to this problem as *"one of the most interesting and useful goals for the mathematical science of computation"* (McCarthy 1963). However, despite a large body of work in the area (Dybjer 1986; Guttman, Ramsdell, and Wand 1995; Moore 1989; Morris 1973; Oliva, Ramsdell, and Wand 1995; Thatcher, Wagner, and Wright 1980; Young 1989), we still lack the technology to prove automatically the correctness of an optimizing compiler. Even manual proofs are rare, and they tend to verify only the algorithms rather than the implementations. Plus, the correctness proofs need to be redone after even the slightest modification or improvement to the compiler.

Proving compiler correctness is just a means towards the actual goal of ensuring that only correct output is ever produced by the compiler. In this paper we propose a potentially more practical approach to the same goal. Instead of verifying the compiler once and for all, we check aspects of the correctness of every individual compilation. This will not ensure that the compiler is bug-free, but it will signal most incorrect compiler outputs as soon as they are produced. To reduce the complexity of the checking process, we do not try to check full equivalence of the source and target programs, but instead we verify only that the target program has certain key properties that can be verified using a small amount of information about the source program.

We present in this paper the design and implementation of Touchstone, an optimizing compiler that translates a strongly typed programming language (essentially a type-safe subset of C) into DEC Alpha assembly language, and a *certifier* that checks the type safety of any assembly language program produced by the compiler. The result of the certifier is either a formal proof of type safety or a counterexample pointing to a potential violation of the type system by the assembly-language target program. We refer to the ensemble of the compiler and the certifier as a *certifying compiler*.

Our approach provides several advantages:

- This method is significantly easier to employ than a formal verification of the compiler, even if the formal verification is restricted to proving that only type-safe code is emitted. This is because it is easier in general to verify the correctness of the result of a computation than to prove the correctness of the computation itself. Furthermore, with this approach, most compiler revisions and improvements do not require any change to the certifier.
- This method can be applied to optimizing compilers, because the design of the certifier does not restrict the

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

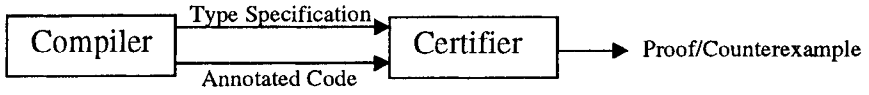


Figure 1: Overview of the Touchstone certifying compiler.

optimizations that the compiler is allowed to perform. Our optimizing compiler generates code that, for many programs, matches or is within 15% of the performance of both gcc and cc with all optimizations enabled, the difference being due mostly to several optimizations that we have not yet implemented. Also, we have successfully tested the certifier on hand-optimized assembly language.

- The presence of the certifier drastically improves the effectiveness of compiler testing because, for each test case, it statically signals compilation errors that might otherwise require many executions to detect. Even though this approach does not ensure full compiler correctness, in our experience the vast majority of compiler bugs lead the compiler to generate unsafe target programs for at least one of the test cases.

- This method is applicable to the compilation of any type-safe language, as well as for certifying other properties of the target programs beyond type safety. Also, a significant benefit of our design is that it requires relatively few modifications to the traditional compiler design, and hence it should be possible to adapt existing compilers to this technique.

- This is a practical method for producing, in an automatic manner, the safety proofs for a Proof-Carrying Code (Necula 1997; Necula and Lee 1996) system for type safety. By attaching the type-safety proof emitted by the certifier to the assembly language program, we enable a circumspect software system to easily verify (by checking that the attached proof is valid and applies to the given target program) that the program is type safe and memory safe. Thus, a certifying compiler can be at the base of a system for safe execution of untrusted mobile code.

This paper is organized as follows. In Section 2 we give a high-level overview of the certifying compiler that we have implemented, and we compare it with related systems. Then we present some details of the source language compiled by our prototype compiler. We continue with the implementation details of the compilation and the certification phases. We discuss the certification phase first (Section 4) because its design is of independent interest and because it sets up the requirements for the compiler subsystem, which is discussed in Section 5. Of all of the optimizations, we focus on array bounds-checking elimination and we show what additional output the compiler must produce so that the certifier can check the memory safety of the optimized code (Section 5.1). We conclude with experimental results on a range of realistic C programs (Section 6). The experiments show that the cost of generating and checking the safety proofs is low, and also that we are indeed certifying a true optimizing compiler whose output code performance approaches that of both cc and gcc.

2 Overview of the Touchstone Certifying Compiler

At a high-level, the certifying compiler is, as shown in Figure 1, a pipeline composed of a compiler and a certifier. The compiler is a traditional compiler adapted to produce type specifications and code annotations in addition to the assembly language target program. Determining whether the target programs are type safe and memory safe is not an easy matter, due to the fact that the compiler performs a wide range of global optimizations. For example, the compiler performs global register allocation (with spilling and coalescing), and so a register might be used to store values of different types within a single code block. Also, the compiler aggressively analyzes and removes array-bounds checks, thus making it nontrivial to deduce that the target code is memory safe. (The full range of optimizations performed by our compiler is described in Section 5.)

The purpose of the code annotations is to make it possible for a simple certifier to understand enough of the code to verify its type safety and memory safety, despite the optimizations. Owing to the design of the certifier, the required annotations are limited to loop invariants that declare the types of the live registers at the beginning of a loop body. The type specifications declare the type of argument and result registers for every function in the code. The type specifications are thus the vehicle for propagating source level information to the certification stage and to allow the certifier to verify that the target program retains at least the typing characteristics of the source program, if not full equivalence.

The certifier subsystem is itself a pipeline composed of three subsystems: the verification condition generator (referred to as *VCGen*), the prover and the proof checker, as shown in Figure 2. The VCGen scans the annotated assembly language program and, using the type specifications and the code annotations, produces a *safety predicate* for each function in the code, such that the safety predicate has a proof if and only if the assembly language program is memory-safe and type-safe according to the typing specification. Due to the code annotations and typing specifications, the VCGen can be performed on a function-at-a-time basis and can be implemented as an efficient single pass through the program.

Following the VCGen phase, the safety predicate is submitted to a prover for first-order predicate logic that produces a formal proof of the predicate. Finally, the safety predicate and its proof are given to a very simple proof checker that verifies that we actually have a valid proof of the required safety predicate, and therefore the compiler output is memory safe and type safe.

An important characteristic of our system is that it has a small safety-critical infrastructure. That is, the code that is relied upon to guarantee that no unsafe target programs escape unnoticed includes only the VCGen and the proof checker. Neither the compiler nor the prover need to be correct in order to be guaranteed to detect incorrect compiler output. This is a significant advantage, since the VC-

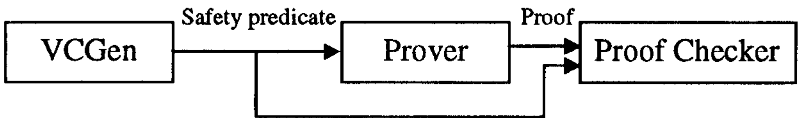


Figure 2: The structure of the certifier.

Gen and proof checker are significantly simpler than the compiler and the prover. Our confidence in VCGen and the proof checker is further enhanced by the fact that they are borrowed unchanged from our Proof-Carrying Code system, (Necula and Lee 1996) which has been in use since September 1996.

3 The Source Language

Our current prototype implementation of a certifying compiler is for a strongly typed language, essentially a type-safe subset of the C programming language. Unlike C, all array subscripting operations are implicitly guarded by bounds-checking conditionals. Also, in order to simplify the elimination of bounds-checking, an array is represented as a pair of values representing the base address and the array length. The `length` operator refers to the length component, while the subscripting operation refers to the base address component. This arrangement is compatible with the common programming practice of passing the array length value together with the base address. Multidimensional arrays have a length component for every dimension.

In addition to safe arrays, our compiler supports Java-style exceptions and exception handling (mostly for a cleaner treatment of array subscript errors), dynamic allocation of data structures in the heap, booleans as a separate type, and most of the arithmetic expression constructs of the C programming language. Common language features that are currently missing are: recursive data structures, function pointers, floating point numbers, and allocation of data structures on the stack. Of these, only the function pointers are expected to pose some difficulties because they are not currently supported by the certification subsystem. And of course, we do not implement casts, the address-of operator, pointer arithmetic and explicit memory deallocation because they are not safe in general (though perhaps safely restricted versions of these operators might be added in the future). Finally, the implementation of the language assumes the use of an automatic garbage collector.

4 Design Details of the Certifier

The design of the certifier establishes the required code annotations and type specifications that the compiler must produce. There are other important aspects of the certifier's design, as well. Although we shall discuss only type and memory safety here, the certifier is general enough to be used for certifying other properties, and for handling safety properties in other languages.

For a more concrete presentation of the certification process we introduce a simple example program and the corresponding compiler output. The program in Figure 3a computes the sum of all elements of an integer array. Our compiler (which compiles one function at a time) compiles this program into the annotated code shown in Figures 3b and the typing specification shown in 3c. Note that the source-level array argument is represented in the target program as

```
int main(int a[]) {
   int i, s = 0;
   for(i=0;i<length(a);i++) {
     s += a[i];
   }
   return s;
}
```

(a)

```
      #a0 - base address, a1 - array length
main: mov  zero, v0           #s=0
      mov  zero, t0           #i=0
L1:   ANN_INV(v0 : int ∧ t0 >= 0,{t0,t1,v0})
      subl   t0,a1,t1         #i - length(a)
      bge    t1,L2
      s4addl t0,a0,t1         #t1=a0+4*t0
      addl   t0,1,t0          #i++
      ldl    t1,0(t1)         #a[i]
      addl   t1,v0,v0
      br     L1
L2:   ret
```

(b)

$$\texttt{main} \; : \; (\texttt{Pre} = a_0 : \texttt{array}(\texttt{int}, a_1) \land a_1 \geq 1, \; \texttt{Post} = v_0 : \texttt{int})$$

(c)

Figure 3: An example source program (a) and the corresponding compiler output, consisting of the annotated code (b) and the typing specification (c).

two values, namely the base address in register a_0 and the array length in register a_1. The return value is returned in register v_0, following the standard DEC Alpha calling convention. Note also that our compiler is successful in removing the bounds-checking operations in this example. The syntax and meaning of the loop invariant code annotation appearing at label L1 in Figure 3b and the typing specifications from Figure 3c are described in the next section.

4.1 The VCGen

The VCGen is a verification condition generator for the DEC Alpha annotated assembly language. Traditionally, verification condition generation is implemented as a backward pass through the code. However, we choose a different implementation technique that uses a forward symbolic-evaluation pass through the code. VCGen operates on a per-function basis and performs three main operations. Firstly, it ensures that the code satisfies certain simple syntactic conditions (e.g., that all branch targets are within the code boundaries and that only recognized instructions occur). Secondly, VCGen evaluates the code symbolically and whenever it encounters a memory operation it emits a verification condition (VC) that states under what conditions the memory operation is considered safe. For example, in the case of a read operation from address a, the condition "$\texttt{saferd}(a)$" is emitted. For a write operation the condition "$\texttt{safewr}(a, e)$"

$$
\begin{array}{lll}
\textit{Vars} & x ::= & \mathbf{r}_m \mid \mathbf{r}_i \qquad (i = 0, \ldots, 31) \\
\textit{Expr} & e ::= & x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid \\
 & & \texttt{sel}\,(e_1, e_2) \mid \texttt{upd}\,(e_1, e_2, e_3) \\
\textit{Types} & \tau ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{array}(\tau, e) \\
\textit{Pred} & P ::= & \texttt{true} \mid P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x.P_x \mid \\
 & & e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \\
 & & e : \tau \mid \texttt{saferd}\,(e_1) \mid \texttt{safewr}\,(e_2, e_3) \\
\textit{Inv} & \iota ::= & \texttt{ANN\_INV}(P, \{x_1, \ldots, x_k\}) \\
\textit{Spec} & \sigma ::= & f : (\texttt{Pre} = P_1, \texttt{Post} = P_2)
\end{array}
$$

Figure 4: The syntax of the safety predicates.

is emitted (e denotes the value being written). The meaning of the predicates `saferd` and `safewr` is defined at the level of the prover (described in Section 4.2) to allow for a greater flexibility in choosing the desired flavor of memory safety.

VCGen must operate on a per-function basis and analyze code with loops without having to iterate through the loop body multiple times. To accomplish this, we require that each function in the code have a typing specification in the form of a precondition and a postcondition, and that each loop have an invariant annotation, which is a predicate that must hold every time around the loop. As these specifications and annotations come with the code and cannot be trusted in general, the third function of VCGen is to ensure that they exist and are valid. To ensure that all loop invariants are present, VCGen verifies that each backward-branch target is associated with an invariant annotation.

In order to set the stage for a more detailed discussion of VCGen, we proceed with the introduction of the required notation. The symbolic evaluator operates with the syntactic entities shown in Figure 4.[1] Among the variables we have the 32 physical DEC Alpha registers ($\mathbf{r}_i, i = 0, \ldots, 31$) and the memory pseudo-register $\mathbf{r}_m$. The latter is used to denote the contents of the memory during execution. The contents of a memory address a is written as $\texttt{sel}(\mathbf{r}_m, a)$ and the effect of updating the memory at address a with the expression e is modeled by the assignment $\mathbf{r}_m \leftarrow \texttt{upd}(\mathbf{r}_m, a, e)$. We write *CS* and *Temp* to refer to the callee-save and temporary machine registers, as defined by the DEC Alpha calling convention.

The language of predicates contains the first-order predicate logic constructors, the memory-safety predicates and the typing predicate. Among the types we consider here only the integers, booleans and one-dimensional arrays. Note that the array type encodes not only the element type but also the array length, which is guaranteed to be at least one. The type of pointers to elements of type τ is expressed as $\texttt{array}(\tau, 1)$.

The only code annotations that we need for the purpose of this paper are the loop invariant annotations. Each such annotation contains an invariant predicate and a set of registers that are modified in the loop body (see the invariant at label `L1` in Figure 3b). For a simpler presentation we show the code annotations as part of the code although in practice they are stored in the data segment.

The typing specification of a function is a pair of a *precondition* and a *postcondition*. The precondition is essentially a description of the calling convention and it declares the type of each argument register used by a function. The

[1]We only show here the syntax that is required for the examples. In practice, a more comprehensive language of expressions and predicates is used.

$$
\begin{array}{l}
\forall \mathbf{a}_0.\ \forall \mathbf{a}_1.\ \forall \mathbf{r}_m. \\
\quad (\mathbf{a}_0 : \texttt{array}(\texttt{int}, \mathbf{a}_1) \wedge \mathbf{a}_1 \geq 1) \supset \\
\qquad (0 : \texttt{int} \wedge 0 \geq 0) \wedge \\
\qquad (\forall \mathbf{t}_0.\forall \mathbf{t}_1.\forall \mathbf{v}_0. \\
\qquad\quad (\mathbf{v}_0 : \texttt{int} \wedge \mathbf{t}_0 \geq 0) \supset \\
\qquad\qquad (\mathbf{t}_0 - \mathbf{a}_1 \geq 0 \supset \mathbf{v}_0 : \texttt{int}) \wedge \\
\qquad\qquad (\mathbf{t}_0 - \mathbf{a}_1 < 0 \supset \\
\qquad\qquad\qquad (\texttt{saferd}(\mathbf{a}_0 + 4 \times \mathbf{t}_0) \wedge \\
\qquad\qquad\qquad \mathbf{t}_0 + 1 \geq 0 \wedge \\
\qquad\qquad\qquad \mathbf{v}_0 + \texttt{sel}(\mathbf{r}_m, \mathbf{a}_0 + 4 \times \mathbf{t}_0) : \texttt{int})))
\end{array}
$$

Figure 6: The safety predicate for the annotated code of Figure 3b.

postcondition is a similar declaration of the types of the result registers ($\mathbf{v}_0$ and $\mathbf{r}_m$ according to the standard calling convention on the DEC Alpha). A function returning no result has the postcondition `true`. The specifications are easily derived from the type of the function (see Figure 3c). Intuitively, the precondition is a predicate that can be assumed to be true when analyzing the body of the function, while the postcondition is a predicate that must be made true by the body of the function.

VCGen is defined as a symbolic evaluator whose result is a predicate (the verification condition) that is provable only if the program is safe with respect to the typing specification. Let $\Sigma : \textit{Label} \rightarrow \textit{Spec}$ be the type specification for the entire program, represented as a map from function labels to their typing specifications. We assume that the target program is an array Π of instructions and code annotations. The state of the symbolic evaluator consists of the current index i in the target program Π, the register state ρ and a list $\mathcal{L}$ of the loop invariants encountered on the path from the start of the function. (Recall that VCGen translates one function at a time.) The register state is a mapping from register names to expressions $\rho \in \textit{VarState} = \textit{Vars} \rightarrow \textit{Expr}$. We write $\rho[\mathbf{r}_i \leftarrow e]$ to denote assigning of e to $\mathbf{r}_i$ and we write $\rho(e)$ to denote the expression obtained after substituting the register names with their values in ρ. We extend the substitution notation to predicates. The loop invariant mapping $\mathcal{L}$ maps the indices of loop invariants to the register states at the beginning of the corresponding loop body. These states are used to verify the set of changed registers in a loop.

The core of VCGen is the symbolic evaluator, which can be described as a function $SE_{\Pi,\Sigma,\rho_0,Post}(i, \rho, \mathcal{L})$ with seven parameters: the annotated program Π, the type specification Σ, the initial register state and the postcondition of the current function (ρ_0 and *Post*), and the current values of the instruction index i, the register state ρ and the loop state $\mathcal{L}$.

To compute the safety predicate of a function f with precondition *Pre* and postcondition *Post*, we first initialize the registers with new variables $x_0, \ldots, x_{32}$ (for the machine registers $\mathbf{r}_0, \ldots, \mathbf{r}_{31}$ and the memory pseudo-register $\mathbf{r}_m$). If ρ_0 is the resulting initial register state, then the safety predicate is given by the formula:

$$SP_f = \forall x_0 \ldots x_{32}.\rho_0(Pre) \supset SE_{\Pi,\Sigma,\rho_0,Post}(f, \rho_0, [])$$

To simplify the notation we omit the subscripts on the *SE* function from now on.

The symbolic evaluation function is defined formally as a recursive function in Figure 5, and described informally in

$SE(i+1, \rho[\mathbf{r}_d \leftarrow \rho(\mathbf{r}_1 + \mathbf{r}_2)], \mathcal{L})$	if	$\Pi_i = \mathtt{addl}\ \mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_d$
$(\rho(\mathbf{r}_s) = 0 \supset SE(i+n+1, \rho, \mathcal{L})) \wedge$ $(\rho(\mathbf{r}_s) \neq 0 \supset SE(i+1, \rho, \mathcal{L}))$	if	$\Pi_i = \mathtt{beq}\ \mathbf{r}_s, n$ $\Pi_{i+n+1} = \mathtt{ANN\_INV}$ if $n < 0$
$\mathtt{saferd}\ (\rho(\mathbf{r}_s) + n) \wedge$ $SE(i+1, \rho[\mathbf{r}_d \leftarrow \rho(\mathtt{sel}(\mathbf{r}_m, \mathbf{r}_s + n))], \mathcal{L})$	if	$\Pi_i = \mathtt{ldl}\ \mathbf{r}_d, n(\mathbf{r}_s)$
$\mathtt{safewr}\ (\rho(\mathbf{r}_d) + n, \rho(\mathbf{r}_s))) \wedge$ $SE(i+1, \rho[\mathbf{r}_m \leftarrow \rho(\mathtt{upd}(\mathbf{r}_m, \mathbf{r}_d + n, \mathbf{r}_s))], \mathcal{L})$	if	$\Pi_i = \mathtt{stl}\ \mathbf{r}_s, n(\mathbf{r}_d)$
$\rho(Pre) \wedge \forall y_1 \ldots y_k . \rho'(Post) \supset SE(i+1, \rho', \mathcal{L})$	if	$\Pi_i = \mathtt{jsr}\ g$ and $\Sigma_g = (Pre, Post)$ $(\rho', \{y_1, \ldots, y_k\}) \leftarrow \mathtt{scramble}(\rho, Temp)$
$\rho(Post) \wedge \mathtt{checkEq}\ (\rho, \rho_0, CS)$	if	$\Pi_i = \mathtt{ret}$
$\rho(P) \wedge \forall y_1 \ldots y_k . \rho'(P) \supset SE(i+1, \rho', \mathcal{L}[i \leftarrow \rho'])$	if	$\Pi_i = \mathtt{ANN\_INV}\ P, s$ and $i \notin Dom(\mathcal{L})$ $(\rho', \{y_1, \ldots, y_k\}) \leftarrow \mathtt{scramble}(\rho, s)$
$\rho(P) \wedge \mathtt{checkEq}\ (\rho, \mathcal{L}_i, Regs - s)$	if	$\Pi_i = \mathtt{ANN\_INV}\ P, s$ and $i \in Dom(\mathcal{L})$

$$\begin{aligned} \mathtt{scramble}(\rho, \{x_1, \ldots, x_k\}) &= (\rho[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k], \{y_1, \ldots, y_k\}) \text{ , } y_i \text{ are new variables} \\ \mathtt{checkEq}(\rho, \rho_0, s) &= \bigwedge_{x \in s} \rho(x) = \rho_0(x) \end{aligned}$$

Figure 5: The definition of the symbolic evaluator function $SE_{\Pi,\Sigma,\rho_0,Post}(i, \rho, \mathcal{L})$. The result of symbolic evaluation is a safety predicate which is true if the program is safe.

the rest of this section. For arithmetic operations the evaluator updates the symbolic register state and continues with the next instruction. In the case of a conditional branch both branches are evaluated, each with the appropriate assumption about the outcome of the conditional. Note the use of implication to include control flow information in the resulting verification condition, so that the verification condition can be proved without further reference to the code. A backward branch is verified to point to an invariant instruction. This is a simple way to verify that all loops have at least one invariant and to ensure the termination of the symbolic evaluator. For a memory operation, the appropriate safety predicate is emitted, in addition to updating the register state.

When dealing with a function call the evaluator must ensure that the precondition part of the typing specification for the target function is established prior to the call. As is the case with the memory operations, VCGen does not itself verify the precondition, but instead it emits the appropriate verification conditions so that the precondition is verified by the prover when proving the verification condition. The symbolic evaluator assumes conservatively that all temporary registers are changed during a function invocation. The unknown effect of the function call on the temporary registers is expressed in the symbolic evaluator with the help of the `scramble` operation, defined at the bottom of Figure 5. To process the code following the function call, the symbolic evaluator uses the register state produced by `scramble` and assumes that the postcondition component of the typing specification is met upon return. Note how quantification on the new values of the temporary registers is used to ensure that they are "new" from the logical point of view.

When the symbolic evaluator encounters the return instruction, it emits verification conditions that are provable only if the current function's postcondition is satisfied and if all the callee-save registers are preserved since the beginning of the function. The latter condition is encoded as a conjunction of equalities between the values of registers on function entry (as encoded by the register state ρ_0) and their values on function exit.

A loop invariant annotation is dealt with in a manner similar to a function call or a return instruction, depending on whether this is the first time it is encountered. When a loop invariant is encountered for the first time, the symbolic evaluator verifies that the invariant is established before the loop is started. Then the symbolic evaluator simulates an arbitrary iteration through the loop, and for that purpose creates new register values for those registers that are declared to be modified by the loop body. In order to process the loop body, the symbolic evaluator uses the new values of registers and assumes that the invariant holds in this new state before the execution of the loop body. A loop invariant that is encountered for the second time marks the end of the arbitrary iteration that was initiated at the first occurrence of the invariant. At this time, the evaluator requires that the invariant be established and that only registers that were declared to be modified by the loop body have actually been modified.

We conclude the presentation of the VCGen by showing in Figure 6 the safety predicate that it produces for the program of Figure 3.

$$\frac{e : \mathtt{array}(\tau, l) \qquad 0 \leq i \qquad i < l}{\mathtt{saferd}(e + 4 \times i)}$$

$$\frac{e : \mathtt{array}(\tau, l) \qquad 0 \leq i \qquad i < l}{\mathtt{sel}(m, e + 4 \times i) : \tau}$$

Figure 7: Proof rules for proving the safety of array accesses. Currently only base types can occur in arrays, thus the size of an array entry is four bytes.

4.2 The Prover and the Proof Checker

To prove the safety predicates produced by VCGen we need a theorem prover for first-order logic. Many of the existing theorem provers (Boyer and Moore 1979; Detlefs 1996; Gordon 1985; Owre, Rushby, and Shankar 1992) can be used for this purpose, although they do not produce proofs that can be checked independently. That is not an impediment as long as we agree to rely on the correctness of the prover, and to give up the possibility of using the certifying compiler as a front end to Proof-Carrying Code systems. However, we feel that these are important properties, and thus, to retain them we have implemented a theorem prover that emits proofs. The theorem prover is based on the Nelson-Oppen architecture for cooperating decision procedures (Nelson and Oppen 1979), also implemented in the Stanford Pascal Verifier (D.C. Luckham 1979) and the Extended Static Checking (Detlefs 1996) systems.

Theorem provers are traditionally viewed as logically-incomplete systems that require human intervention in many instances. In our system, however, *the theorem prover is guaranteed to be able to prove the safety predicates automatically* because these predicates are implicitly proved by the compiler itself during compilation.

For example, during bounds-checking elimination, the compiler eliminates those bounds-checking conditionals that it can prove to be always true. Later, during certification, the corresponding array operation prompts the symbolic evaluator to emit a predicate that captures exactly the arithmetic facts that were proved by the compiler. Thus, it is enough for the theorem prover to be "as good" at proving arithmetic facts as the compiler is. This is usually the case in practice, as theorem provers are much more powerful than the typical compiler analysis of arithmetic.

Beyond the predicate calculus and simple linear arithmetic, the theorem prover must also be able to interpret the typing and the memory-safety predicates that occur in the symbolic evaluator's output. This can be done in most theorem provers by specifying a collection of inference rules. Two such rules are shown in Figure 7. The first rule says that it is safe to read an element of an array if its index is within the array boundaries, and the second rule says that the result of this read operation has the type of the array elements. By using these rules plus the usual predicate calculus rules, the reader can verify informally that the safety predicate shown in Figure 6 is indeed valid, and therefore the assembly language program of Figure 3b is memory safe.

The role of the proof checker is to verify that every step in the proof is valid and also that the proof proves the required safety predicate and not another one. We use the proof checker of the Proof-Carrying Code system, which represents proofs in a language based on LF (Harper, Honsell, and Plotkin 1993), a simple typed λ-calculus. There are several engineering advantages of using LF to represent proofs, perhaps the most fundamental being that proof checking can be accomplished simply by type checking of LF terms. We encode a proof as an LF expression and the safety predicate as an LF type. Then LF type-checking is enough to validate the proof. (The fact that this approach is sound is established in (Harper, Honsell, and Plotkin 1993). We have made some modifications that are described and proved to be sound in (Necula and Lee 1997).)

Another advantage of this arrangement is that the LF type checker is independent of the particular logic, and thus we are able to reuse its implementation for checking proofs in many logics, including the memory-safety and type-safety logic presented here. Also LF and LF type checking are simple, which leads to a small and fast implementation of the proof checker.

5 The Optimizing Compiler

The compiler component of our system is not very different from a traditional compiler for C. The differences can be classified as due to changes in the language semantics and due to changes in the requirements on the output. The former class includes the enforcement of the array bounds, as mentioned before. The latter class includes the mechanisms for emitting the code annotations and type specifications.

A common task in producing both the loop invariants and the type specifications is the conversion of variable type declarations to typing predicates involving machine registers. This is done in two stages. The first stage happens in the compiler front-end and consists of generating a predicate $t : \tau$ for every source-level variable v of type τ, where t is the intermediate language temporary variable corresponding to v. Because we have chosen the type components of predicates to be similar to the source-level types, this stage is very simple. The second stage is done after register allocation and consists of replacing the temporaries occurring in predicates with the register names chosen for them by the allocator.

The procedure described above is all that is necessary for producing the type specifications. For loop invariants, we have to emit typing predicates for the variables that are live at the beginning of the loop body, and we also have to compute the set of registers that are changed in the loop body. This is done by a separate pass over the output program.

One of the goals of the compiler implementation is to show that even the output of an optimizing compiler can be certified for type-safety. The main optimizations that we have implemented are: array bounds-checking elimination, constant propagation with algebraic reductions, dead-code elimination, common-subexpression elimination, loop invariant hoisting, in-register global variables, induction variable elimination, and global register allocation. Most of the implementation effort was directed towards array bounds-checking elimination both because bounds-checking is our most significant handicap with respect to the C compilers compiling the same programs, and because it is notoriously difficult to verify the memory safety of assembly language programs whose bounds-checking code was eliminated. Our results in this area are a major advantage over TIL (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996) and Java (Gosling, Joy, and Steele 1996) bytecode verification.

The type-safety aspect of the certification is always insensitive to most optimizations that a compiler might perform, including all of the above. This is not true for the memory-safety aspect of the certification. The most obvious complication for memory safety is generated by array bounds-checking elimination. The only other optimization implemented in our compiler that complicates the certification of memory-safety is the induction variable elimination in the instance when it replaces the array indexing with a running pointer inside the array. We discuss here only the array bounds-checking elimination.

5.1 Array Bounds-Checking Elimination

The array bounds-checking elimination is implemented in our compiler as an instance of the more general conditional elimination, that is, the elimination of the conditionals whose boolean expression can be statically proved to be always true or always false. The proof is attempted using a simple decision procedure for linear arithmetic based on computing loop residues (Shostak 1981).

The conditional elimination analysis is implemented as a pass through the intermediate representation. When a bounds-checking conditional is encountered, its boolean expression is converted to the form $x - y + c \geq 0$, where x and y are arbitrary expressions (usually variables) and c is a constant. This form is submitted to the loop residue decision procedure that returns a value saying that, in the current state, the boolean is always true, or always false, or that its value cannot be determined statically. In the first two cases the conditional is replaced with the code of the appropriate branch, otherwise the boolean expression is recorded in the decision procedure's state and the "true" branch is considered recursively. When the true branch is finished, the boolean is retracted and its negation is asserted instead for processing the "false" branch. Because all conditionals involved in array bounds-checking are of the form $x \geq 0$ or $x < y$, and because the loop residue is complete for this fragment of arithmetic, our compiler is able, in practice, to eliminate almost all bounds checks.

There are two situations when the above analysis does not succeed in eliminating bounds-checks. One is when the information required for the proof is external to the current function. This happens, for example, in the function

```
int sub(int a[], int i) {return a[i];}
```

because there is no way to verify statically that i is a valid index for a. This situation would not occur if the function were inlined at the call site.

To cover for the lack of interprocedural analysis, we have extended the language to allow the programmer to write simple function preconditions consisting of boolean expressions involving the formal parameters. For example, to eliminate the bounds check in the above function the programmer can write:

```
int sub(int a[], int i)
PRECONDITION (0 <= i && i < length(a)) {
    return a[i];
}
```

The function preconditions are assumed true when analyzing the function but are checked at the call site. The preconditions are a convenient way to hoist the bounds checks out of the function to the call site, where there might be more information for eliminating them. In our experiments, these checks are in most cases eliminated by the same conditional elimination phase that eliminates the array bounds checks.

Another situation when the conditional elimination analysis presented above might fail to eliminate bounds-checks is inside loops like the one in Figure 3a. In that example, the upper bound of the index is given by the loop termination conditional, while the lower bound is implicit. It can be seen from the loop invariant in that example that the compiler discovers a lower bound ($t_0 \geq 0$) and emits it as part of the invariant. To deal with such situations the compiler first discovers *monotone variables*. A variable v is monotone if, on all paths through the loop body, it is incremented by expressions that are either all positive or all negative. To detect monotone variables, the compiler first collects a set of increments for each variable, and then using the same loop-residue decision procedure verifies the sign of the set elements. For a monotone variable with only positive increments, the compiler generates a loop invariant stating that the value of the variable is always greater or equal than the value of the same variable on loop entry. This is how the conjunct $t_0 \geq 0$ appeared in the invariant annotation of Figure 3b.

6 Experimental Results

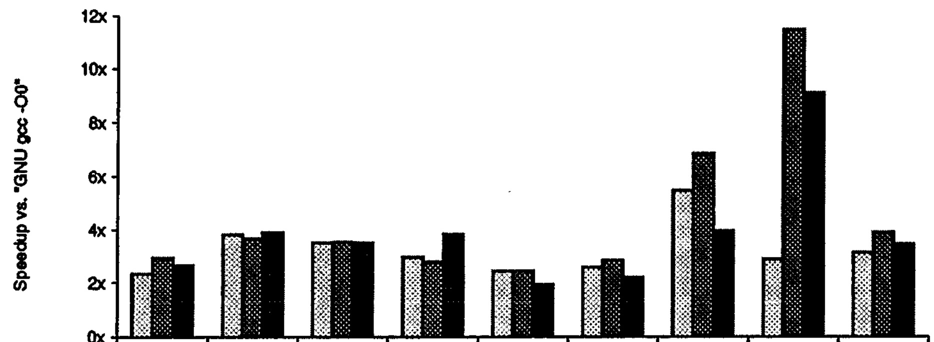
We have two purposes in reporting the results of our experiments with the Touchstone certifying compiler. First, we wish to support the claim that we are applying the certification technique to an optimizing compiler. And second, we wish to show that the costs of certification are reasonably low. For the first purpose, we compare the running times of several benchmarks compiled by Touchstone against the running times of the same programs compiled with the GNU `gcc` compiler and the vendor-supplied compiler (DEC `cc`) with all optimizations enabled. For the second purpose, we measure the size of proofs and also the time consumed for VC generation, theorem proving, and proof checking. We compare these with the code size and the compilation time respectively.

Our benchmark programs depend only on those language features that are currently implemented in the certifying compiler (this ruled out floating-point benchmarks, for example) with a bias towards programs for which array-bounds checking elimination could make a significant difference in the running time. We furthermore preferred programs that might be useful as native-code components in a safe mobile code system, in order to evaluate the certifying compiler as a front-end to a system for safe execution of Proof-Carrying Code.

These considerations led us to eight benchmarks. Three of them, `blur`, `sharpen`, and `edge` are bidimensional convolutions used as image processing filters in the `xv` program. `qsort` is an implementation of the quicksort algorithm for an array of integers. `simplex` is the linear programming algorithm implemented for rational numbers. `kmp` (an implementation of the KMP search algorithm) and `unpack` (one of the `gzip` decompression algorithms and the core of the Unix utility with the same name) were chosen as examples of cases where array bounds-checking elimination is not effective. The `bcopy` program is an implementation of string copy for non-overlapping strings. It is worth noting that some of these C programs are fairly realistic in both size and complexity, and none required anything more than minor syntactic modifications to conform to our safe C dialect. The main changes involved replacing the use of pointer arithmetic with array indexing. All results are the average of at least 1000 runs on a DEC Alpha 21064 running at 175MHz.

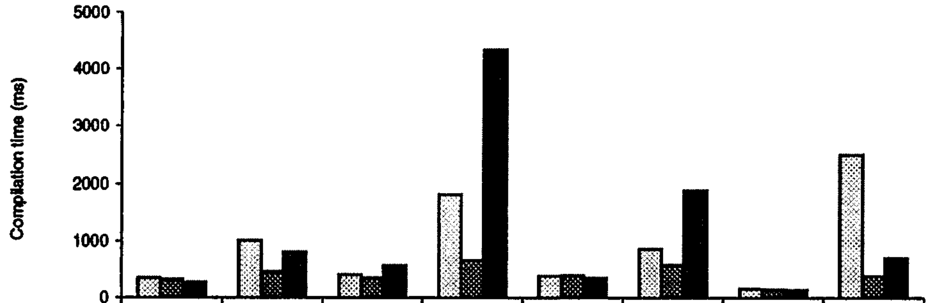
Figure 8 shows the effect of optimizations on the running time of the benchmark programs for the GNU `gcc` compiler, the DEC `cc` compiler, and the certifying compiler. The C compilers were invoked with all optimizations enabled (`-O4`). The running times are reported as speedups over the running time of the unoptimized code as compiled with `gcc -O0`. The last set of bars in Figure 8 is the geometric mean of the speedups for each compiler. On the average, the certifying compiler performs slightly better than `gcc` (by about 10%) and not quite as well as `cc` (the difference being about 12%). The programs for which the certifying compiler is not quite as good as the C compilers are `kmp` and `unpack`, due to the bounds checks that cannot be eliminated, and `bcopy`, because of the lack of loop-unrolling in the certifying compiler. In addition to array bounds-checking elimination, the inter-procedural register allocation and the common-subexpression elimination played a major role in making the quality of code generated by Touchstone comparable to that produced by the other C compilers.

In our experiments, the C compilers compile the programs unsafely (that is, without any bounds checking), while Touchstone has the handicap of having to implement (and



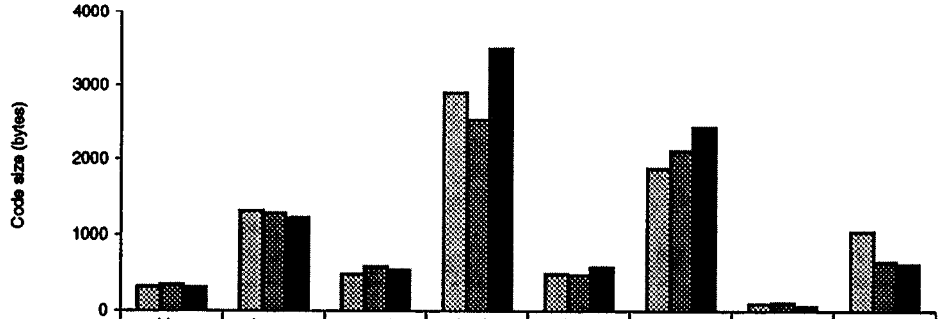
	blur	sharpen	qsort	simplex	kmp	unpack	bcopy	edge	GMEAN
▨ GNU gcc -O4	2.33	3.82	3.51	2.97	2.44	2.59	5.50	2.92	3.17
▩ DEC cc -O4	2.92	3.68	3.52	2.79	2.44	2.84	6.88	11.52	3.94
■ Touchstone	2.64	3.89	3.52	3.86	1.93	2.20	4.00	9.16	3.48

Figure 8: The effect of optimizations in the certifying compiler, expressed as the ratio between the running time of the optimized code to the running time of the same code compiled with "GNU gcc -O0". For comparison, we also show for each benchmark the effect of the optimizations in the GNU C compiler ("GNU gcc -O4") and the vendor C compiler ("DEC cc -O4"). The last column is the geometric mean over all the benchmarks.



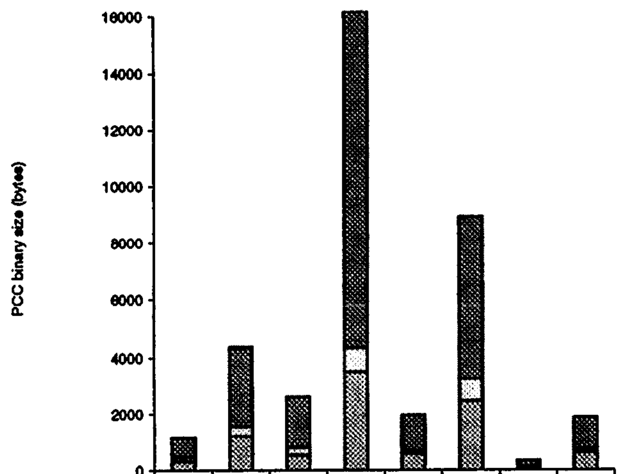
	blur	sharpen	qsort	simplex	kmp	unpack	bcopy	edge
▨ GNU gcc -O4	350	1010	410	1810	380	870	150	2500
▩ DEC cc -O4	325	460	340	650	400	580	145	380
■ Touchstone	271.0	818.0	560.0	4340.0	348.0	1885.0	136.0	697.0

Figure 9: Comparison of the compilation time for Touchstone and the GNU gcc and DEC cc compilers with all optimizations enabled. The times in the table are shown in milliseconds. On the average, Touchstone is 20% slower than gcc and 72% slower than cc. Note that the compilation time does not include VC generation, proof generation or proof checking.



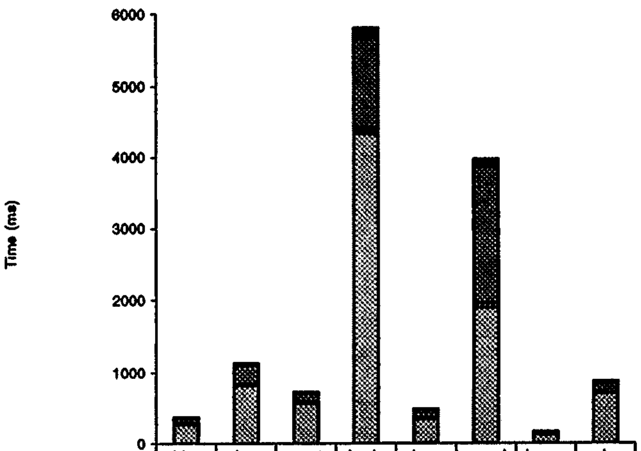
	blur	sharpen	qsort	simplex	kmp	unpack	bcopy	edge
▨ GNU gcc -O4	320	1312	480	2896	480	1872	96	1040
▩ DEC cc -O4	336	1280	592	2528	464	2112	112	656
■ Touchstone	304	1216	544	3488	576	2432	64	624

Figure 10: Comparison of the target code sizes for programs compiled with Touchstone and the GNU gcc and DEC cc compilers with all optimizations enabled. The sizes in the table are shown in bytes of machine code. On the average, Touchstone is within 5% of the sizes of code emitted by the C compilers.



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
▩ Proof (bytes)	718	2774	1778	11810	1234	5636	250	1122
▢ Annot (bytes)	162	342	272	825	132	804	36	102
▨ Code (bytes)	304	1216	544	3488	576	2432	64	624

Figure 11: The relative sizes (in bytes) of proofs, invariants and the machine code.



	blur	sharp	qsort	simplex	kmp	unpack	bcopy	edge
■ Proof Checking (ms)	7.7	23.5	16.3	120.3	9.6	92.5	3.6	14.7
▩ Proving (ms)	81.0	257.0	127.0	1272.0	108.0	1912.0	25.0	143.0
■ VC Generation (ms)	6.3	20.9	11.5	73.9	8.4	70.6	3.5	9.7
▨ Code Generation (ms)	271.0	818.0	560.0	4340.0	348.0	1865.0	136.0	697.0

Figure 12: The distribution of time spent for the compilation and certification of several benchmarks. The data in the table is expressed in milliseconds.

then hopefully remove) the array-bounds checks. The array-bounds checking elimination described in Section 5.1 is able to eliminate most of those checks whose proof is local to the current function, but is ineffective when the elimination requires global information. This weakness is a problem in all of our benchmarks except for `blur`, `edge` and `bcopy`. To substitute for the required global information in these cases, we have added simple one-line function preconditions to `sharpen`, `qsort` and `simplex`. With the preconditions, our compiler succeeds in eliminating all bounds-checking operations in all but the kmp and unpack benchmarks. What makes these two benchmarks special is that array indices are computed based on the contents of some auxiliary data structures. The formal safety argument for these array operations involves the proof of complicated global program invariants, and thus it is probably not reasonable to expect a compiler to be able to eliminate these bounds checks.

Even though the preconditions are added to programs only for the benefit of the bounds-checking elimination in our compiler, we do not feel that this gives us an unfair advantage over the C compilers. To the contrary, the preconditions enable more extensive bounds-checking elimination and thus make the job of the certifier more difficult. The formal proof of redundancy for the bounds-checks that are eliminated based on preconditions and global information is larger and more complicated than for the locally-provable checks. Our experiments show that the additional bounds-checking elimination that is enabled by the preconditions leads, on the average, to a 7% reduction in code size, a 12% reduction of the running time and a 12% increase of the proof sizes.

Due to the fact that Touchstone is an early prototype, the compilation time is significantly larger than that of the C compilers used in the performance comparisons. Figure 9 shows the compilation times (not including the time for VC generation, proof generation or proof checking) of Touchstone and of the C compilers (with all optimizations enabled) for our set of benchmarks. On the average, Touchstone is 20% slower than GNU gcc and 72% slower than DEC cc. Figure 10 shows the comparison of the machine-code sizes of programs compiled with Touchstone and the C compilers. Unlike the compilation times, the sizes of machine code emitted by Touchstone are within 5% of that emitted by the C compilers. Note, however, that there is no fundamental reason why a certifying compiler should emit code that is larger than that emitted by a traditional compiler. With respect to the compilation time, the certifying compiler must incur the extra cost of emitting the loop invariants and type specifications. This cost, however, should negligible with respect to the rest of the compilation effort.

Hoping to have convinced the reader that we are indeed certifying optimized assembly language, we now move to the presentation of the costs of certification. For this purpose, we have measured the proof size and the time required for VC generation, theorem proving and proof checking, for the benchmarks discussed above.

Figure 11 shows the sizes of the safety proofs and the annotations as compared to the sizes of the machine code for each benchmark. The annotations are only 30% of the size of the code, on the average. The average ratio of proof size to code size is 2.5, which is consistent with our observations in experiments with PCC using hand-written assembly language. While this factor seems large, one must consider that the proofs are not currently compressed. Preliminary measurements show that general-purpose compression algorithms can decrease the size of proofs by a factor of two. However, larger reductions are likely to be obtained by first optimizing the proof representations and then employing a compression algorithm. Further discussion about proof optimizations is given in Section 8.

Figure 12 displays graphically the distribution of time spent for compilation and certification. On the average, 72% of the time is spent compiling, 22% is used for theorem proving and the rest of 6% is split evenly between VC generation and proof checking. Based on these results we make two observations. First, the cost of certification is only about a third of the cost of compilation, meaning that it is reasonable to use the certifier throughout the life of the compiler, and not just during compiler development. Second, not only are VCGen and the proof checker much simpler than the compiler and the theorem prover, but they are also much faster. Hence, this safety-critical infrastructure is both small and fast. This is important in situations when the certifying compiler is used to produce Proof-Carrying Code, because the system receiving the code needs to trust and run only the VCGen and the proof checker.

7 Related Work

The idea of checking individual compilations instead of verifying the compiler also appears in the work of Cimatti et al. (Cimatti et al. 1997), though in the much simpler instance of a non-optimizing compiler from an expression language without loops or function calls to an RTL-like language. On the other hand they have the more ambitious goal of verifying full equivalence of the source expression and the target program.

The compilation approach presented here resembles in many respects the compilation strategy of the TIL (Tarditi, Morrisett, Cheng, Stone, Harper, and Lee 1996) compiler for Standard ML, which uses a typed intermediate language that can be easily type-checked to achieve an independent validation of optimizations. However, the TIL type-system does not guarantee memory safety in the presence of certain optimizations such as array bounds-checking elimination, and furthermore, it cannot be used after the register allocation phase when some variables (registers) are reused to hold values of different types in the body of the same function. For this reason, types are dropped in TIL before the register allocation phase and thus, no type-checking is possible at the level of the compiler output. The problems related to register allocation are solved by Morrisett et al. (Morrisett, Walker, and Crary 1998) by choosing a more expressive type system, but the issue of memory-safety in the presence of optimizations such as array bounds-checking elimination still remains a problem.

The purpose and the design of our certifying compiler are also related to the Java (Gosling, Joy, and Steele 1996) compiler and bytecode verifier (Lindholm and Yellin 1997) systems. The similarity is that both systems produce code that is annotated for the purpose of enabling a certification system (the bytecode verifier, in the Java case) to verify the type safety. The difference is that our certifier has a more flexible annotation language that permits the verification of arbitrarily optimized assembly language while necessitating fewer annotations. The bytecode verifier only works on a specially designed bytecode intermediate language where typing annotations are contained in the instruction codes themselves. Furthermore, the Java bytecode verifier prevents the compiler from doing several important optimizations, such as array bounds-checking elimination and global register allocation, since these checks are built in to the definition of the byte codes.

8 Discussion and Future Work

The approach to a certifying compiler presented in this paper is inspired by our work on Proof-Carrying Code (PCC) (Necula 1997; Necula and Lee 1996), and in fact reuses the VCGen, the theorem prover, and the proof checker components of our implementation of PCC. If integration with PCC and the generality and simplicity of the certifier were not important to us, we could have chosen from several alternate implementation approaches.

One alternative is suggested by the fact that the verification conditions emitted by VCGen can be proved automatically. Thus, one can incorporate parts of the prover in VCGen and prove the VCs as they are encountered, without actually generating a safety predicate and maybe not even a proof that can be checked. This might be particularly practical when array bounds are not verified and thus only a small part of the prover is used. The Java bytecode verifier (Lindholm and Yellin 1997) can be viewed as taking this approach, as can the type-checker in the typed assembly language of Morrisett, et al. (Morrisett, Walker, and Crary 1998).

Another variation of the method presented here is to attempt to certify the output of an off-the-shelf compiler, which does not produce annotations or type specifications. We suspect that this can be achieved by interposing a loop invariant inference phase before VCGen. For the source language presented here, and for a compiler that does not perform aggressive global optimizations, it should be possible in principle to discover the typing invariants completely automatically.

Our current experimental results show that the proofs are about 2.5 times larger than the code. Some preliminary experiments show that standard compression techniques reduce the proof sizes by a factor of 2. We believe, however, that the biggest gains in reduction of size will be obtained by designing and implementing optimizations in the representation of proofs. Already, our current LF representation provides a simple approach to type reconstruction that allows some type information to be elided (Necula and Lee 1997). We are currently exploring more aggressive techniques that involve finding common subterms (essentially a kind of common subexpression elimination). A manual inspection of the proofs gives some indication that such an approach should yield good reductions, though much further work is necessary to measure the effects.

In addition to the general notion of certifying compilation, we believe that we have discovered a simple correctness criterion for both register allocation with spilling and for instruction scheduling. Bugs in these compiler optimizations are notoriously difficult to find because they lead to subtle errors in the output that tend to surface as sporadic program failures, usually many instructions past the actual erroneous instruction. Furthermore, the low-level nature of the output and the fact that such errors most likely occur in large programs, makes the visual inspection of the output quite tedious.

We have observed that the result of our symbolic evaluator is insensitive to global register allocation with spilling and to global code scheduling. During the development of our compiler, this has meant that we could verify each run of these transformations simply by comparing the safety predicates computed before and after the transformation. To see an example of this, consider the annotated code of Figure 3b. In this code, the register t_1 is used to hold values of different types in the body of the loop. One can now observe that even if the independent uses of r_1 are renamed, the safety predicate does not change, up to the renaming of bound variables in the predicate. A similar experiment shows the same phenomenon in the case of instruction scheduling.

To preserve this invariance property even in the presence of register spilling, the symbolic evaluator must be extended to interpret a portion of the stack frame as an extension of the register file and thus consider the read/write operations to the stack frame as moves from/to these pseudo-registers. To simplify the symbolic evaluator, only memory references to addresses that are computed as immediate offsets from the dedicated stack pointer (or frame pointer) register are intercepted; all other memory references must be proved to be within heap-allocated arrays.

We strongly suspect that this observation can form the basis for a general correctness criterion for global register allocation and instruction scheduling, and one that would be useful for any compiler, not only certifying compilers. However, we do not yet have a formal proof of this claim. We hope to make a more formal statement and proof of this correctness criterion in future work.

9 Conclusion

This paper presents the design and implementation of a certifying compiler composed of a traditional optimizing compiler for a typed language and a certifier that automatically produces a proof of type safety for each assembly language program resulting from the compilation. The main benefit of such a system over a traditional compiler is that the certifier acts as an effective referee for the correctness of each compilation, thus simplifying compiler testing and development. Only rare compilation errors that do not break the type-safety of the target program are not detected by a certifying compiler. During the development of the certifying compiler we have encountered only one such error, as opposed to a large number of errors that were caught early by the certifier. The certifier reduced the effort required for the development of an optimizing compiler whose performance rivals that of production compilers, to only three man-months.

A second important benefit of a certifying compiler is that it can serve as an automatic front-end to a system that uses Proof-Carrying Code to enable the safe execution of untrusted mobile code.

The main contribution of this research is the design of a certifier that does not restrict the optimizations that the compiler can perform, while requiring only a small amount of information from the compiler. As an indirect result, we have identified that the symbolic evaluation technique that is at the base of the certifier leads to a simple but effective correctness criterion for low-level optimizations such as register allocation and code scheduling.

Acknowledgments

The authors would like to thank Gary Lindstrom and Trevor Jim for the helpful comments and suggestions on earlier drafts of this paper. We also thank the anonymous referees for many suggestions on how to improve this paper.

References

Boyer, R. and J. S. Moore (1979). *A Computational Logic.* Academic Press.

Cimatti, A. et al. (1997, June). A provably correct embedded verifier for the certification of safety critical software. In *Computer Aided Verification. 9th International Conference. Proceedings*, pp. 202–213. Springer-Verlag.

D.C. Luckham, e. (1979, March). Stanford Pascal verifier user manual. Technical Report STAN-CS-79-731, Dept. of Computer Science, Stanford Univ.

Detlefs, D. (1996). An overview of the Extended Static Checking system. In *Proceedings of the First Formal Methods in Software Practice Workshop.*

Dybjer, P. (1986). Using domain algebras to prove the correctness of a compiler. *Lecture Notes in Computer Science* (182).

Gordon, M. (1985, July). HOL: A machine oriented formulation of higher-order logic. Technical Report 85, University of Cambridge, Computer Laboratory.

Gosling, J., B. Joy, and G. L. Steele (1996). *The Java Language Specification.* The Java Series. Reading, MA, USA: Addison-Wesley.

Guttman, J. D., J. D. Ramsdell, and M. Wand (1995). VLISP: a verified implementation of Scheme. *Lisp and Symbolic Computation* (8), 5–32.

Harper, R., F. Honsell, and G. Plotkin (1993, January). A framework for defining logics. *Journal of the Association for Computing Machinery 40*(1), 143–184.

Lindholm, T. and F. Yellin (1997, January). *The Java Virtual Machine Specification.* The Java Series. Reading, MA, USA: Addison-Wesley.

McCarthy, J. (1963). Towards a mathematical theory of computation. In C. M. Popplewell (Ed.), *Proceedings of the International Congress on Information Processing*, pp. 21–28. North-Holland.

Moore, J. S. (1989). A mechanically verified language implementation. *Journal of Automated Reasoning* (5), 461–492.

Morris, F. L. (1973). Advice on structuring compilers and proving them correct. In *Proceedings of the First ACM Symposium on Principles of Programming Languages*, pp. 144–152.

Morrisett, G., D. Walker, and K. Crary (1998, January). From System F to typed assembly language. In *The 25th Annual ACM Symposium on Principles of Programming Languages.* ACM. To appear.

Necula, G. C. (1997, January). Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pp. 106–119. ACM.

Necula, G. C. and P. Lee (1996, October). Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations*, pp. 229–243. Usenix.

Necula, G. C. and P. Lee (1997, October). Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, Computer Science Department, Carnegie Mellon University.

Nelson, G. and D. Oppen (1979, October). Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*(2), 245–257.

Oliva, D. P., J. D. Ramsdell, and M. Wand (1995). The VLISP verified PreScheme compiler. *Lisp and Symbolic Computation* (8), 111–182.

Owre, S., J. M. Rushby, and N. Shankar (1992, June). PVS: A prototype verification system. In D. Kapur (Ed.), *11th International Conference on Automated*

Deduction (CADE), Volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga, NY, pp. 748–752. Springer-Verlag.

Shostak, R. (1981, October). Deciding linear inequalities by computing loop residues. *Journal of the ACM 28*(4), 769–779.

Tarditi, D., J. G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee (1996, May). TIL: A type-directed optimizing compiler for ML. In *PLDI'96 Conference on Programming Language Design and Implementation*, pp. 181–192.

Thatcher, J. W., E. G. Wagner, and J. B. Wright (1980). More on advice on structuring compilers and proving them correct. *LNCS'94: Proceedings of a Workshop on Semantics-Directed Compiler Generation* (94), 165–188.

Young, W. D. (1989). A mechanically verified code generator. *Journal of Automated Reasoning* (5), 493–518.