Automatic Detection and Diagnosis of Faults in Generated Code for Procedure Calls

Mark W. Bailey
Jack W. Davidson

TR-2001-1

May 2001

Department of Computer Science
Hamilton College
Clinton, NY 13323

AUTOMATIC DETECTION AND DIAGNOSIS OF FAULTS IN GENERATED CODE FOR PROCEDURE CALLS

Mark W. Bailey
Hamilton College

Jack W. Davidson
University of Virginia

Abstract

In this paper we present a compiler testing technique that closes the gap between existing compiler implementations and correct compilers. Using formal specifications of procedure calling conventions, we have built a target-sensitive test suite generator that builds test cases for a specific aspect of compiler code generators: the procedure calling sequence generator. By exercising compilers with these specification-derived target-specific test suites, our automated testing tool has exposed bugs in every compiler tested. These compilers include some that have been in heavy use for many years. Once a fault has been detected, the system can often suggest the nature of the problem. The testing system is an invaluable tool for detecting, isolating, and correcting faults in today's compilers.

1 Introduction

Building compilers that generate correct code is difficult. To achieve this goal, compiler writers rely on automated compiler building tools and thorough testing. Automated tools, such as parser generators, take a specification of a task and generate implementations that are more robust than hand-coded implementations. Conversely, testing tries to make hand-coded implementations more robust by detecting errors. One aspect of a compiler that has traditionally been hand-coded is the portion that generates *calling sequences*—implementations of procedure calls. We have developed a language, called CCL, for specifying procedure calling conventions. CCL specifications are used to automatically generate calling sequences for the *vpcc/vpo* retargetable optimizing compiler [1]. While experimenting with CCL, we realized that the descriptions could be used to make other compilers more robust without requiring that the compiler implementation use CCL. In this paper, we describe how CCL's

underlying model can be used to generate tests for hand-coded calling sequence generators in other compilers. This technique has exposed a number of calling convention errors in production-quality compilers that have been heavily used for years.

In this paper we describe several contributions. First, we present a method for automatically testing implementations of procedure calling conventions. Using this technique we have found bugs in mature **C** compilers. This approach, which uses a formal model of procedure calling conventions, methodically generates tests that offer complete coverage of the problem domain. Second, we introduce an algorithm for intelligently selecting important tests from the complete coverage suite. These tests include boundary cases that are more likely to reveal bugs than exhaustive or randomly generated tests. Third, because the tests focus only on the calling convention, they isolate errors more effectively than tests from a general test suite. Fourth, we describe a method for automatically diagnosing the nature of some types of faults. Finally, we describe a method for quickly determining the conformance of multiple compilers at once.

2 Procedure Calling Conventions

An important feature of high-level programming languages that compilers must implement is the procedure call. The interface between procedures facilitates separate compilation of program modules and interoperability of programming languages. This is accomplished by defining a *procedure calling convention* that dictates the way that program values are communicated and how machine resources are shared between a procedure making a call (the *caller*) and the procedure being called (the *callee*). The calling convention is machine-dependent because the rules for passing values from one procedure to another depend on machine-specific features such as memory alignment restrictions and register usage conventions. The code that implements the calling convention, known as the *calling sequence* [2], must be generated by the code generator. This aspect of the code generator, which we name the *calling sequence generator*, is a source of great difficulty for the compiler writer because it not only suffers from being hand-coded, it also changes each time the compiler is retargeted.

2.1 A Simple Calling Convention

To aid in our discussion of calling conventions, we use a simplified example calling convention. Figure 1 contains the calling convention rules for a hypothetical machine. Consider the following ANSI **C** prototype for a function `warp`:

```
int warp(char p1, int p2, int p3, double p4);
```

For the purpose of transmitting procedure arguments for our simple convention, we only consider the *signature* of the procedure. We define a procedure's signature to be the procedure's name, the order and types of its arguments, and its return type. This is analogous to ANSI **C**'s *abstract declarator* [3], which for the previous function prototype is:

```
int warp(char, int, int, double);
```

which defines a function that takes four arguments (a `char`, two `int`'s, and a `double`), and returns an `int`.

1. Registers R1, R2, R3, and R4 are 32-bit argument-transmitting registers.
2. Arguments are also passed on the stack in increasing memory locations starting at the stack pointer.
3. An argument may have type `char` (1 byte), `int` (4 bytes), or `double` (8 bytes).
4. An argument is passed in registers (if enough are available to hold the entire argument), and then on the stack.
5. Arguments of type `int` are 4-byte aligned on the stack.
6. Arguments of type `double` are 8-byte aligned on the stack.
7. Stack elements that are skipped over cannot be allocated later.
8. Return values are passed in registers R1 and R2.
9. Values of registers R6, R7, R8, and R9 must be preserved across a procedure call.

Figure 1. Rules for a simple calling convention.

With `warp`'s signature, we can apply the calling convention in Figure 1 to determine how to call `warp`. Arguments to `warp` would be placed in the following locations:

- `p1` in register R1,

- `p2` in register R2,
- `p3` in register R3, and
- `p4` on the stack at offsets 0–7.

Notice that although register R4 is available, `p4` is placed on the stack since it cannot be placed completely in argument-transmitting registers (rule 4). Such restrictions are common in actual calling conventions.

2.2 Convention to Implementation

Once the calling convention has been established, a compiler can be targeted to generate the calling sequence code that implements the procedure calls for the source language. Traditionally, this code has been hand crafted. In contrast, we use a calling convention specification and an interpreter. The interpreter can generate tables that can be used in the calling-convention-specific portion of *vpcc/vpo* [1], or in a test suite generator. The test suite generator uses information from the table to tailor the test suite to the specific calling convention. The test suite can either be used to confirm that the *vpcc/vpo* implementation properly uses the convention tables, or that another, independent compiler conforms to the convention described in the CCL specification. In the next section, we describe the formalism that the CCL interpreter uses to generate the convention tables.

3 The Formal Model

We use finite state automata to model a calling convention's placement of arguments (and return values) a machine's memory locations. The use of FSA's for modeling parts of a compiler, and as an implementation tool, has a long and successful history. For example, FSA's have often been used to implement lexical analyzers [4]. More recently, Proebsting and Fraser [5], and Müller [6] have used finite state automata to model and detect structural hazards in pipelines for instruction scheduling.

3.1 P-FSA Representation

An example FSA that we use to model calling convention placement is shown in Figure 2. This FSA models the placement of procedure arguments for the simple calling convention described previously. A placement FSA (P-FSA) takes a procedure's signature as input and produces locations for the procedure's arguments as output. The automaton works by moving from state to state as the location of

each value is determined. When a transition is used to move from one state to the next, information about the current parameter is read from the input, and the resulting location is written to the output.

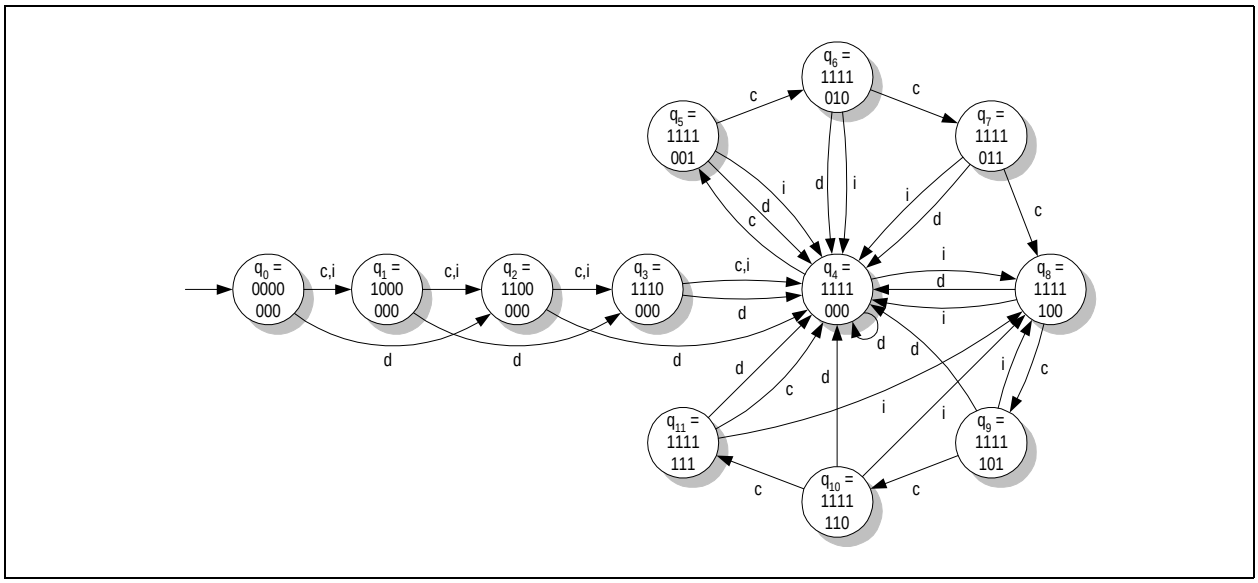

Figure 2. P-FSA for transmission of parameters for a simple calling convention.

The states of the machine represent the state of allocation for the machine's memory resources. For example, the state q_2 (labeled 1100 000), represents the fact that registers R1 and R2 have been allocated (the first two bits: 11), that registers R3 and R4 have not been allocated (the second two bits: 00), and that the stack pointer is currently eight-byte aligned (the remaining three bits: 000). A transition between states represents the placement of a single argument. Since arguments of different types and sizes impose different demands on the machine's resources, we may find more than one transition leaving a particular state. In our example, q_8 has three transitions even though two of them (`int` and `double`) have the same target state (q_4). This duplication is required since the output from mapping an `int` is different from the output from mapping a `double`.

In the CCL specification language, the stack is modeled as an infinite, or boundless resource. Modeling the allocation of an infinite resource using an FSA poses a problem, however. As mentioned previously, the state represents which resources have been allocated. For finite resources, this is easily accomplished by maintaining a bit vector. When a resource no longer may be used, the associated bit is set to indicate this. For an infinite resource this scheme cannot work if we hope to use an FSA since

this would require a bit vector of infinite length. To simplify the problem, we impose a restriction on infinite resources: their allocation must be contiguous. Thus, for an infinite resource $I = \{i_1, i_2, \ldots\}$, we can store the allocation state by maintaining an index p whose value corresponds to the index of the first available resource in I. Because the allocation of I must be contiguous, p partitions the resources since a resource i_j is unavailable if $j < p$ or available if $j \geq p$. For instance, if the stack is the infinite resource, p can be considered the stack pointer.

Nevertheless, we still have a problem. Although for a particular machine, the value of p must be finite, the resulting FSA could have as many as 2^{32} stack allocation states for a 32-bit machine. However, we can significantly reduce this number by observing that the decision of where to place a parameter in memory is not based on p, but rather on alignment restrictions. For our example, we care only if the next available memory location is one-, four-, or eight-byte aligned. Consequently, we can capture the allocation state of the machine with three bits that distinguish the memory allocation states. We call these the *distinguishing* bits for infinite resource allocation.

Handling pass-by-value structures creates an analogous problem. Structures of different sizes allocate different amounts of space. Hence, each structure of a different size impacts the state of resource allocation differently. This implies that each P-FSA state requires an infinite number of exiting transitions; one for every different structure size. Fortunately, since only the "alignment state" of the stack pointer is of interest, we need only include transitions for structures that leave the P-FSA in a different state. So for a convention that requires structures to be passed in 8-byte aligned memory locations, all structures of size n where $n \bmod 8 = 1$ share the same transition out of a given state because they leave the alignment, p, in the same state. Therefore, the number of transitions leaving a state is limited by the alignment restrictions of the machine.

Placement functions are described in terms of finite resources, infinite resources, and selection criteria. A set of finite resources $R = \{r_1, r_2, \ldots, r_n\}$ is used to represent machine registers, while an infinite resource $I = \{i_1, i_2, \ldots\}$[1] is used to represent the stack. The *selection criteria*

1. This can easily be extended to model more than one infinite resource.

$C = \{c_1, c_2, \ldots, c_m\}$ correspond to characteristics about arguments (such as their type and size) that the calling convention uses to select the appropriate location for a value. We encode the signature of a procedure with a tuple $w \in (C^*, C^*)$. The first element of the tuple contains zero or more[1] return value criteria, while the second element contains zero or more parameter criteria. Each state q in the automaton is labeled according to the allocation state that it represents. The label includes a bit vector v of size n that encodes the allocation of each of the finite resources in R. Additionally, to express the state of allocation for the stack, we include d, the distinguishing bits that indicate the state of stack alignment. So, a state label is a string vd that indicates the resource allocation state. In our example convention, $n = 4$, and the length of the string d ($|d|$) is 3. So, each state is labeled by a string from the language $\{0, 1\}^4\{0, 1\}^3$. The output of the machine is a string $s \in P$, where

$$P = R \cup \{0, 1\}^{|d|}$$

which contains the placement information.

Since the P-FSA produces output on transitions, we have a Mealy machine [7]. We define a P-FSA, M, as a six-tuple[2] $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where:

- Q is the set of states with labels $\{0,1\}^n\{0,1\}^{|d|}$ representing the allocation state of machine resources,
- the input alphabet $\Sigma = C$, is the set of selection criteria,
- the output alphabet $\Delta = P$, is the set of memory location strings,
- the transition function $\delta: Q \times \Sigma \rightarrow Q$,
- the output function $\lambda: Q \times \Sigma \rightarrow \Delta^+$, and
- q_0 is the state labeled by $0^n w$ where $|w| = |d|$, and w is the initial state of d.

1. Supports languages that allow multiple return values.
2. We use the notation of Hopcroft and Ullman for finite state automata and regular expressions [19]. We use letters early in the alphabet (a, b, c) to denote single symbols. Letters late in the alphabet (w, x, y, z) will denote strings of symbols.

We also define $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ and $\hat{\lambda}: Q \times \Sigma^* \rightarrow \Delta^*$ which are just string versions[1] of δ and λ, respectively. So, for our example, we have

$$M = (Q, \{\texttt{char}, \texttt{int}, \texttt{double}\}, \{\text{R1, R2, R3, R4}\} \cup \{0, 1\}^3, \delta, \lambda, q_0)$$

where Q and δ are shown in Figure 2 and λ is defined in Table I. Note that we have modified the traditional definition of λ to allow multiple symbols to be output on a single transition. This reflects the fact that arguments can be located in more than one resource. For example, in state q_5 on an `int`, Table I indicates that M produces the string of four symbols 100 101 110 111 that designates four bytes that are four-byte aligned, but are not eight-byte aligned.

λ	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}
char	R1	R2	R3	R4	000	001	010	011	100	101	110	111
int	R1	R2	R3	R4	m_1[a]	m_2[b]	m_2	m_2	m_2	m_1	m_1	m_1
double	R1 R2	R2 R3	R3 R4	m_3[c]	m_3	m_3	m_3	m_3	m_3	m_3	m_3	m_3

Table I. Definition of λ for example P-FSA.

a. m_1 = 000 001 010 011
b. m_2 = 100 101 110 111
c. m_3 = 000 001 010 011 100 101 110 111

The signature:

```
int f(double, double, char, int);
```

will take the P-FSA in Figure 2 along the path $q_0 \rightarrow q_2 \rightarrow q_4 \rightarrow q_5 \rightarrow q_4$ producing the string (R1 R2) (R3 R4) (000) (100 101 110 111) along the way. The parentheses in the output string are required to determine where the placement of one argument ends and the next argument's placement begins. From the string, we can derive the placement of `f`'s arguments. The first `double` is placed in registers R1 and R2, the second in registers R3 and R4, the `char` at the stack location with offset zero and the `int` at the stack location with offset four.

1. Defined by Hopcroft and Ullman [19].

3.2 Completeness and Consistency in P-FSA's

In our experience, we have encountered many recurring difficulties in the calling convention code generators of optimizing compilers for RISC machines[1]. There are three sources for these problems: the convention specification, the convention implementation, and the implementation process. We address each of these in the following paragraphs.

Many problems arise from the method of convention specification. Often, no specification exists at all. Instead the native compiler uses a convention that must be extracted by reverse engineering. In the cases where a specification exists, it typically takes the form of written prose, or a few general rules (*e.g.*, our example description in Figure 1). Such methods of specification have obvious deficiencies. Furthermore, even if we have an accurate method for specifying a convention, it still may be possible to describe conventions that are internally inconsistent, or incomplete. For example, the convention may require that more than one procedure argument be placed in a particular resource. Another possibility is that the specification may omit rules for a particular data type, or combination of data types.

Those problems that do not arise from the specification result from incorrect implementation of the convention. Many of the same problems in the specification process also plague the implementation. Many conventions have numerous rules and exceptions that must be reflected in the implementation. Another difficulty is the implementation may require the use of the convention in several different locations. Maintaining a correspondence between the various implementations can itself be a great source of errors. Finally, this problem is exacerbated by the fact that the implementation frequently undergoes incremental development. Rather than taking on the chore of implementing the entire convention at once, a single aspect of the convention, such as providing support for a single data type, is tackled. After successfully implementing this subset, the next increment is undertaken. During this process, some aspect of the first stage may break due to the interactions between the two pieces.

1. Unlike many CISC machines, RISC machines typically increase the pressure on calling conventions by requiring that procedures pass parameters in registers wherever possible.

The result of these observations is that there are several properties that we would like to ensure about a specification and implementation. The preceding discussion motivates the following categories of questions:

- Completeness:
  - — *Does the specified convention handle any number of arguments?*
  - — *Does the convention handle any combination of argument types?*
- Consistency:
  - — *Does the convention map more than one argument to a single machine resource?*
  - — *Do the caller and callee's implementations agree on the convention?*

Many questions like these can be answered using P-FSA's. The following sections show how we can prove certain properties about P-FSA's that ensure desirable responses to the preceding questions.

3.2.1 Completeness

The completeness properties address how well the convention covers the possible input cases. A convention must handle any procedure signature. If we could guarantee that the convention was complete, or covered the input set, then we could answer the completeness questions posed in the previous section. We can determine if a convention is complete by looking at the resulting P-FSA. For example, will the convention work for any combination of argument types? The answer lies in the P-FSA transitions. For the convention to be complete, each state $q \in Q$ must have $\delta(q, c)$ defined for all $c \in C$. Should there exist some state $q \in Q$ and criteria $c \in C$ such that $\delta(q, c)$ is undefined, then having arrived in state Q on input w, the machine would fail to accept any input string whose prefix was wc. Thus, there would be some signature whose placement could not be determined by the P-FSA. Since all correct P-FSA's must accept all strings in C^*, we can easily detect any P-FSA that implements an incomplete convention by looking for states with missing transitions.

3.2.2 Consistency

The consistency properties address whether the convention is internally and externally consistent. A convention is internally consistent if there is no machine resource that can be assigned to more than one argument. A convention is externally consistent if the caller and callee agree on the locations of transmitted values. In our model, we *detect* internal inconsistency, and *prevent* external inconsistency.

To detect internal inconsistencies, we again turn to the P-FSA. If the convention only used finite resources, detecting a cycle in the P-FSA would be sufficient to detect the error. However, when infinite resources are introduced, so are cycles. We cannot have an internal inconsistency for an infinite resource since p is defined to be monotonically increasing. We detect finite resource inconsistencies in the following manner. An inconsistency can occur when there is a transition from some state q_j to q_k where bit i in the finite bit vector is 1 in q_j, but 0 in q_k. At this point, M has lost the information that resource r_i was already allocated. We can detect this change by comparing all pairs of bit vectors v_1, v_2 such that v_1 labels q_j, v_2 labels q_k and $\delta(q_j, c) = q_k$ for some $c \in C$. To do the comparison, we compute

$$v_3 = (v_1 \oplus v_2) \wedge v_1$$

$v_1 \oplus v_2$ selects all bits that differ between v_1 and v_2. We logically AND this with v_1 to determine if any set bits change value. Thus, if v_3 has any bit set, we have an inconsistency.

Our convention specification language prevents external inconsistencies in the calling convention. A convention specification only defines the argument transmission locations once. Although both the caller and the callee must make use of this information, the specification does not duplicate the information. Since we only have a single definition of argument locations, we only construct a single P-FSA to model the placement mapping. This single P-FSA is used in both the caller and callee. Thus, we prevent external inconsistencies by requiring the caller and callee use the same implementation for the placement mapping.

4 Construction of Diagnostic Programs

Using P-FSA's as an implementation foundation for a compiler enables all of the static analyses described in the previous section. However, when a compiler does not use a P-FSA in its implementation, we can still leverage off the P-FSA formalism to increase the implementation's robustness through systematic testing.

4.1 Test Vector Selection

To test a compiler's implementation of a calling convention, we must select a set of programs to compile and run. To exercise the calling convention, each test program must contain a caller and a callee procedure. For the purpose of testing the proper transmission of program values between procedures, the signature of the callee uniquely identifies a test case. Thus, two different programs whose callees' signatures match perform the same test. Therefore, the problem of generating test cases reduces to the problem of selecting signatures to test.

Selecting which procedure signatures to test is a difficult problem. Because the set of signatures, $S = \{(C^*, C^*)\}$, is infinite one cannot test all signatures. However, since we can model the function that computes the placement of arguments as an FSA, there must be a finite number of states in an implementation to be tested. This is the case for any implementation, including those that do not explicitly use FSA's to model the placement function.

The problem of confirming that an implementation properly places procedure arguments is equivalent to experimentally determining if the implementation behaves as described by the P-FSA state table. This problem is known as the *checking experiment problem* from finite-automata theory [8, 9]. There are numerous approaches to this problem, most of which are based on transition testing. Transition testing forces the implementation to undergo all the transitions that are specified in the specification FSA.

An obvious first approach to generating test vectors using the P-FSA specification is to generate all vectors whose paths through the FSA are acyclic and those whose path ends in a cycle[1]. This solution insures that each state q is visited, and each transition $\delta(q, a)$ is traversed. For an FSA with few states, and a small input alphabet, this may be acceptable. However, the number of such paths for an FSA is $O(|\Sigma|^{|Q|})$. To illustrate the characteristics of P-FSA's, Table II contains profiles for five P-FSA's that we have built from CCL descriptions. For complex conventions, like the MIPS and SPARC, the number of transitions, and more importantly, the number of states can be large. For the

1. We define a *path that ends in a cycle* to be a cyclic path wa where the path w is acyclic.

MIPS, this results in an upper bound of $25^{12} = 2.3\times10^{22}$ test vectors. In practice, the number of test vectors is closer to 10^8 vectors. However, this is still too many to run feasibly.

Machine	Allocation Vector Bits	Memory Partition Bits	$\|Q\|$	$\|\delta\|$	$\|\Sigma\|$	Longest Acyclic Path
DEC VAX	0	0	1	3	3	0
M68020 (Sun)	0	2	4	24	6	3
SPARC (Sun)	6	3	9	90	10	8
M88100 (Motorola)	8	3	72	720	10	15
MIPS R3000 (DEC)	6	3	70	772	25	11

Table II. P-FSA profiles for several calling conventions.

A simpler approach is to guarantee that each transition is exercised at least once. Since there are no more than $|Q| \cdot |\Sigma|$ transitions, the number of test vectors that this generates is not unreasonable. However, this method results in poor coverage that does not inspire confidence in the test suite. For example, for the P-FSA in Figure 2, the three signatures:

```
void f(double, double);
void f(int, int, int, int);
void f(int, double);
```

cover all `int` and `double` transitions leaving states q_{0-2}. This leaves the signature:

```
void f(double, int);
```

untested. Clearly such a test should be included in the suite. To further illustrate the problem, consider the FSA specification shown in Figure 3(a). An erroneous implementation, shown in Figure 3(b), contains an extra state q_1' that is reached on initial input `b`. The two strings, `aaa` and `bbb` completely cover the specification FSA transitions. Unfortunately, these test vectors will not detect that the implementation has an additional (fault) state. Thus, it is not sufficient to include only test vectors that cover the transition set.

An alternative, which falls between the simple transition approach and the acyclic path approach, we call the *transition-pairing* approach. In transition pairing, we examine each state in the

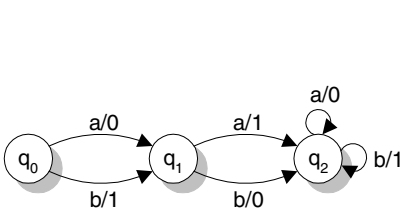

Figure 3(a). Specification FSA.

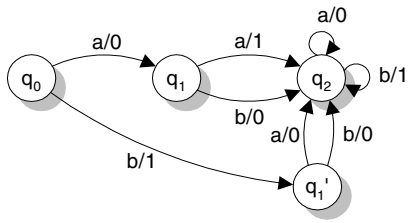

Figure 3(b). Implementation FSA.

Figure 3. Example FSA where a fault will not be detected.

specification FSA. As shown in Figure 4, a state has entering and exiting transitions. For each state, we include a test vector that covers each *pair* of entering and exiting transitions. This eliminates the faulty state detection problem illustrated in Figure 3. To illustrate how, consider the test vectors this process generates: While examining state q_1, transition-pairing will add the substrings aa, ab, ba, and bb to the set of substrings used to generate test vectors. Since the context that these substrings are be used is q_0, they contribute prefixes to the test vector set. Upon exercising q_1 using the prefix ba, the implementation FSA will generate incorrect output: 10 instead of 11. This difference can be identified, and the faulty state detected.

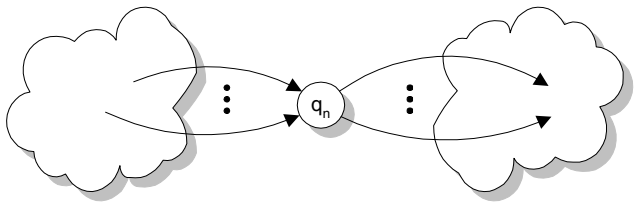

Figure 4. Entering and exiting transitions for a state.

In addition to such fault detection, transition-pairing provides tests that have a similar characteristic to the acyclic method: transitions are tested in all the contexts that they can be applied. Although there are many combinations that are not tested, they are similar to ones included in the set. For example, in the simple FSA pictured in Figure 2, we could have a set of test vectors that includes the vector `double double double` to exercise the state q_4 with the transition pair ((q_2, `double`),

(q_4, `double`)). Such a set would not need to include `int int double double` to cover the same transition pair.

This method of test vector generation provides a complete coverage of transitions in the specification FSA. Further, the tests reflect the context sensitivity that transitions have. This allows for some erroneous state and transition detection, while significantly reducing the number of test vectors. The test vector sizes are significantly smaller than the acyclic method, while still providing a significant degree of confidence.

Machine	Transition Paths	Transition-Pair Paths	Acyclic Paths
DEC VAX	3	12	3
M68020 (Sun)	24	324	96
SPARC (Sun)	224	7,434	$> 10^8$
M88100 (Motorola)	720	22,412	$> 10^8$
MIPS R3000 (DEC)	772	5,655	8×10^8

Table III. Sizes of test suites for various selection methods.

An algorithm for generating transition-pair paths is shown in Figure 5. The algorithm performs a depth-first search of the FSA state graph. Each time a transition (q, a) is encountered, it is marked. This mark indicates that all paths that go beyond (q, a) have been visited. When the algorithm reaches a state q_n on transition (q_m, a), each transition (q_n, b) where $b \in \Sigma$ is visited whether or not it is marked. This causes all pairs of transitions ((q_m, a), (q_n, b)) to be included. These pairs represent all combinations of one entering transition with all exiting transitions. Because the algorithm is depth-first, each entering transition is guaranteed to be visited. Thus, all combinations of entering and exiting transitions are included.

Work related to the automatic generation of test suites has received much attention recently in the area of conformance testing of network protocols [10]. The purpose of these suites is to determine if the implementation of a communication protocol adheres to the protocol's specification. Often, the protocol specification is provided as a finite-state machine. This has resulted in many methods of test selection including the Transition tour, Partial W-method [11], Distinguishing Sequence Method [9],

Input. A finite-state machine M.

Output. The set of transition-pair paths in M that take M from q_0 to q_n with at most one cycle. The set traverses all pairs of transitions $((q_r, a), (q_s, b))$ such that $\delta(q_r, a) = q_s$.

Initial call. TRANSITION-PAIRS(q_0, ε, ∅, 0);

Algorithm:

```
function TRANSITION-PAIRS(q, w, V, cycle)
    paths ← ∅;
    for each a where a ∈ Σ ∧ δ(q, a) is defined do        // For each transition from state q...
        if cycle ≠ 1 ∧ (q, a) ∉ T then                    // No cycles and (q, a) is new
            if q ∉ V then                                // If there is no cycle
                T ← T ∪ {(q, a)};                        // Mark transition as followed
                cycle ← 0;                               // Indicate no cycle
            else
                cycle ← 1;                               // Indicate cycle
            end if
            P ← TRANSITION-PAIRS(δ(q, a), wa, V ∪ {q}, cycle);   // Compute paths from here
            paths ← paths ∪ P;
        end if
        paths ← paths ∪ {wa};                            // Add this path to paths
    end for
    return paths;                                        // Return paths from q
end function
```

Figure 5. Test vector generation algorithm.

and Unique-Input-Output method [12]. These methods are derivatives of the checking experiment problem where an implementation is checked against a specification FSM [13]. Such techniques have also been used in the automatic verification of digital circuits [8, 14]

What distinguishes these methods from ours are the underlying assumptions concerning the characteristics of the implementation FSA's. Unlike theirs, our FSA's can have a large number of states and transitions. This significantly changes the nature of the solution to the problem. Furthermore, much of the problem that network conformance researchers are faced with is identifying which state the implementation FSA is in. A significant portion of their work focuses on generating test vectors that discover the state of the machine. Fortunately, we can always put our implementation FSA in the start state. Also, in their work, a bound on the number of states in the implementation FSA's is assumed. Because we have no practical bound on the number of states in the implementation, their work is not applicable.

4.2 Test Case Generation

After selecting the appropriate test vectors, or procedure signatures, the corresponding test cases must be realized. In our approach, we generate a separate test program for each test vector so that we can easily match any reported errors to the specific test vector.

A procedure call is composed of two pieces: the procedure call within the caller (the call site) and the body of the callee. Because they are implemented differently, these two pieces of code are typically generated in separate locations in a compiler. This natural separation is reflected in the way that we construct our test cases. Each test case is comprised of two files, one contains the caller, the other contains the callee. The two files are compiled and linked together. The programs are self-checking, so that if a procedure call fails, this event is reported by the test itself.

Figure 6 shows the compiler conformance test process. One file is compiled by the compiler-under-test (CUT), while the other is compiled by the reference compiler. The reference compiler operationally defines the procedure calling convention (its implementation is defined to be correct). The resulting object files are linked together and run. Results of the test are checked by the conformance verifier and given to the test conductor. The test conductor tallies the results of all tests for a test suite and generates a conformance report. Although this process uses two compilers, the same process may still be used if a reference compiler is not available. However, this will weaken the conformance verifier's ability to automatically diagnose errors as discussed in the next section.

Figure 7 shows an example test case for the **C** signature `void (int, double, struct(2)`[1]`)`. The caller loads each argument with randomly selected bytes. However, the values of these bytes have an important property: each contiguous set of two bytes is unique. Thus, for a string B of m bytes, for all indexes $0 < i \leq m$, there exists no index $0 < j \leq m$ and $j \neq i$ such that $B[j + k] = B[i + k]$ for all $0 \leq k < 2$. We can easily guarantee this property for all strings B whose length is no more than 65536 (2^{16}) bytes. Since the likelihood of using an argument list of size greater than 64 Kbytes is small, this is sufficient to guarantee that any two bytes passed between procedures are unique. This makes it easier to identify if an argument has been shifted or misplaced. The callee

1. We denote a structure whose size is n bytes as `struct(n)`.

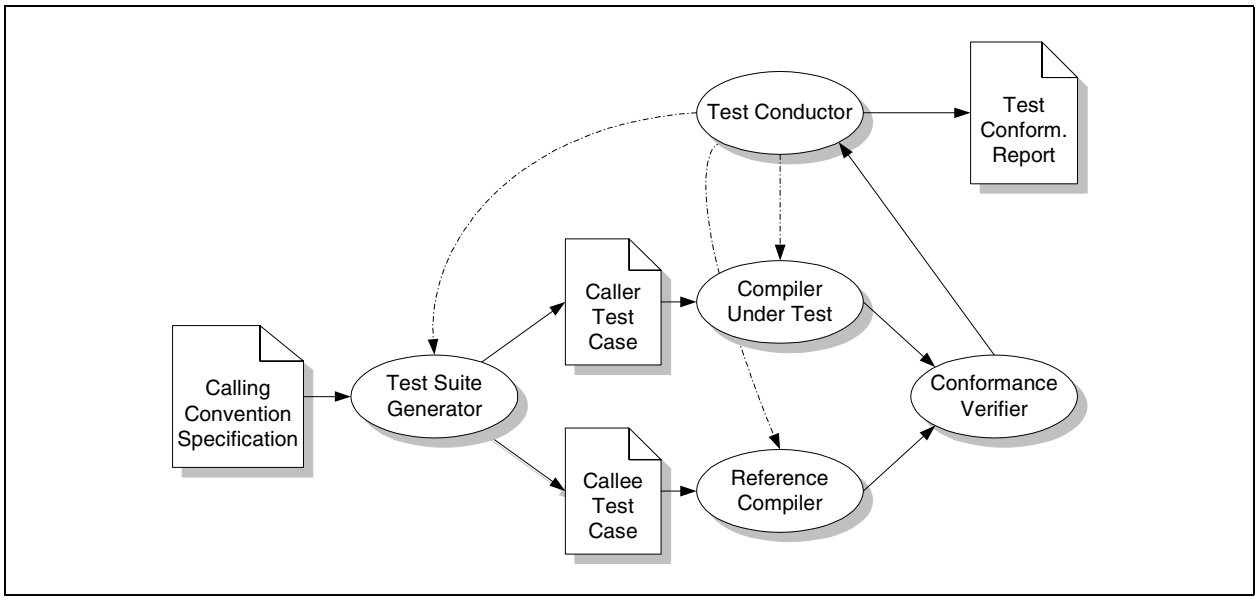

Figure 6. The compiler conformance test process.

receives the values, and checks them against the expected values. If the values do not match, an error condition is signalled.

As one might expect, the generation of good test cases from selected signatures is language dependent. One convention used in the **C** programming language is *varargs*. *varargs* is a standard for writing procedures that accept variable length argument lists. The proper implementation of *varargs* in a **C** compiler is difficult. For each test case that we generate we also generate a *varargs* version to verify that this standard convention is implemented correctly.

4.3 Automatic Diagnosis of Errors

Generation of good tests is only a part of the testing process. If a test fails, the problem must be diagnosed and a solution developed. In this section, we discuss how the second step, diagnosis, can be partially automated.

As discussed earlier, the conformance verifier links a caller and callee together and runs the resulting program. When both a reference compiler and CUT are used, this results in four distinct caller-callee pairs. We call the result of running all four programs an *outcome*. Figure 8 shows an outcome graphically. Procedures generated by the reference compiler are filled circles, while CUT generated components are unfilled. The result of a single test is indicated by an arrow connecting a pair of

```
typedef union {
  unsigned char bytes[8];
  double dbl;
} dblcvt;

typedef struct struct_2 {
  unsigned char field[2];
} struct_2;

void test_function_callee();

void test_function_caller()
{
  dblcvt cvt;
  static struct_2 struct_2_arg_3 = {
    { 0x34,0x8f,}
  };
  double double_arg_2;

  cvt.bytes[0] = 0xe2;
  cvt.bytes[1] = 0xed;
  cvt.bytes[2] = 0xab;
  cvt.bytes[3] = 0xad;
  cvt.bytes[4] = 0x67;
  cvt.bytes[5] = 0x31;
  cvt.bytes[6] = 0xee;
  cvt.bytes[7] = 0x7;
  double_arg_2 = cvt.dbl;

  test_function_callee(0x440260971,
                       double_arg_2,
                       struct_2_arg_3);
}
```

Figure 7(a). Code generated for caller.

```
typedef union {
  unsigned char bytes[8];
  double dbl;
} dblcvt;

typedef struct struct_2 {
  unsigned char field[2];
} struct_2;

void test_function_callee(long_arg_1,
     double_arg_2, struct_2_arg_3)
  long long_arg_1;
  double double_arg_2;
  struct_2 struct_2_arg_3;
{
  dblcvt cvt;

  if(long_arg_1 != 0x440260971) {
    fprintf(stderr, "Bad long_arg_1\n");
    exit(1);
  }
  cvt.bytes[0] = 0xe2;
  cvt.bytes[1] = 0xed;
  cvt.bytes[2] = 0xab;
  cvt.bytes[3] = 0xad;
  cvt.bytes[4] = 0x67;
  cvt.bytes[5] = 0x31;
  cvt.bytes[6] = 0xee;
  cvt.bytes[7] = 0x7;
  if(double_arg_2 != cvt.dbl) {
    fprintf(stderr, "Bad double_arg_2\n");
    exit(1);
  }
  if(struct_2_arg_3.field[0] != 0x34) {
    fprintf(stderr,
"Element 0 is bad in struct_2_arg_3.field\n");
    exit(1);
  }
  if(struct_2_arg_3.field[1] != 0x8f) {
    fprintf(stderr,
"Element 1 is bad in struct_2_arg_3.field\n");
    exit(1);
  }
}
```

Figure 7(b). Code generated for callee.

Figure 7. Example test case.

components. When the result is that a test passed, a solid line is shown, while a dotted line is used for test failure.

The result of a single test, taken in isolation, provides limited information: whether a fault has been detected or not. However, we can glean more information by considering the composite result that an outcome provides. By using multiple versions of object files generated by different compilers, we can exploit the interface of the procedure call. Each test has an object file in common with two

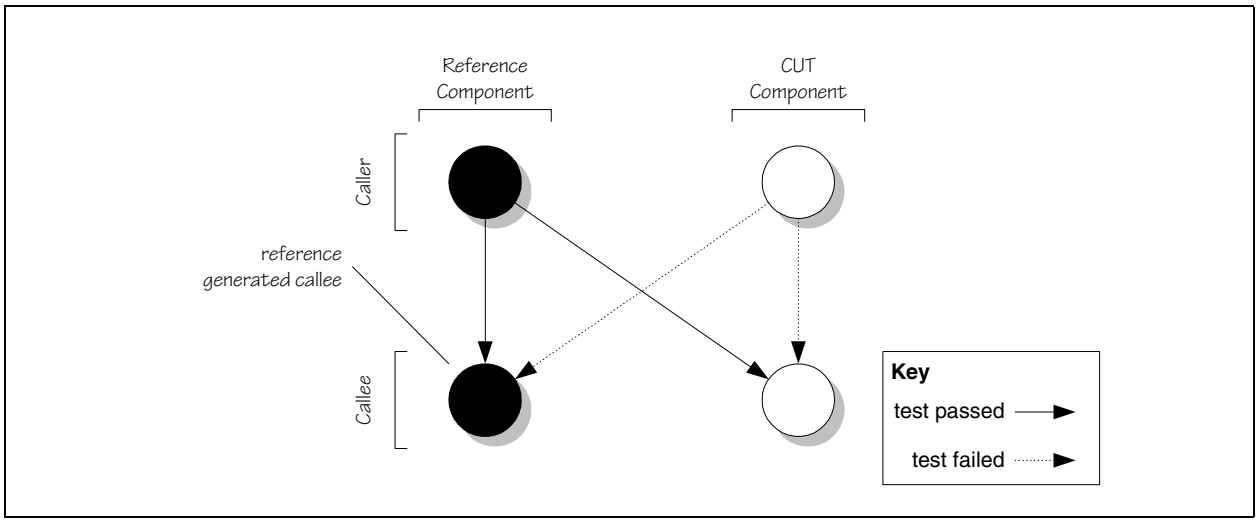

Figure 8. An example outcome.

other tests. When a test fails, the results of the two other tests can help isolate the fault. For example, in the outcome shown in Figure 8, the CUT/reference test (the test comprised of the CUT caller and reference callee) has failed. To isolate if the caller or callee contains the fault, the reference/reference test result is considered. This test replaces the CUT caller with the reference caller, keeping the callee in common between the two tests. Since the test passed, we have reason to believe that the CUT caller contains the fault since the fault disappeared when the CUT caller was removed. Our suspicion is confirmed when we consider the CUT/CUT test. Since this test also failed, the fault remains when the reference callee was removed. Thus, the fault must be in the CUT caller. We would come to the same conclusion had we started with the CUT/CUT fault and considered the CUT/reference and reference/CUT test results.

This method of isolating errors by swapping different components makes it possible to automatically diagnose common errors. Since each outcome is comprised of four results that may indicate a pass or fail, there are 16 outcome configurations. Since this number is small, each outcome can be hand-analyzed once and the results tabulated. Table IV summarizes such an analysis. Several diagnoses deserve mention. First, although the reference compiler is considered the authority, there are many cases where the reference can be determined to be faulty. This occurs in six of the outcomes. Second, three of the outcome configurations are not possible. These are the outcomes where only a single test failed. This indicates a conflict in conventions. This cannot occur with a single test failure

since we assume each component uses a single convention[1]. Finally, for two of the cases, we not only can isolate the location of the fault, but we can identify the nature of the error. This occurs in outcomes D and M where two conflicting conventions have been discovered.

The combination of test vector selection and automatic diagnosis proves to be a powerful debugging tool. As tests are generated, run, and analyzed, patterns of errors tend to emerge. We have found that the patterns themselves suggest the nature of the problem. For example, finding that an error occurred for every signature that included a struct of size greater than seven bytes might suggest an alignment problem. More complicated patterns can exist, and, with knowledge of the calling convention can significantly help the developer correct faults.

4.4 Test Results

We used our technique for selecting test vectors to test several compilers on several target machines. Several errors were found in **C** compilers on the MIPS. In this section, we present these results.

We selected several **C** compilers that generate code for the MIPS architecture (a DECStation Model 5000/125). These included the native compiler supplied by DEC, two versions of Fraser and Hanson's *lcc* compiler [15, 16], several versions of GNU's *gcc* [17], and a previous version of our own **C** compiler, *vpcc/vpo*, that used a hand-coded calling sequence generator [18]. Although we feel that this technique is extremely valuable throughout the compiler development cycle, we believe that it would be fairest to evaluate its effectiveness in finding errors in young implementations of compilers. Where possible, we have used early versions of these compilers. These versions, called *legacy* compilers, represent younger implementations that more accurately exhibit bugs found in initial releases of compilers. However, each of these compilers is a production-quality compiler that has been widely used for years. Finding any bugs in their implementations is still a significant challenge.

---

1. Appel observes that such outcomes actually are possible [20]. In his counter example, the CUT caller implements a different convention than the reference compiler, but the CUT callee implements *both* conventions. In this scenario, the fault is detected in the CUT/reference test, but not in either the CUT/CUT or the reference/reference tests. Although such a case is possible, the chances of a callee implementing two different conventions that do not conflict (*i.e.*, use the same register for two different purposes) are remote. The benefits, in terms of diagnostic ability, of considering such a case as invalid, far outweigh any accuracy gained by labeling it a valid outcome. Finally, if such a case were to occur, it would still be detected; it just could not be automatically diagnosed.

Outcome	Diagnosis	Outcome	Diagnosis
	Outcome A: Faults in at least three components.		Outcome B: Faults in both components of the CUT.
	Outcome C: Faults in both components of reference compiler.		Outcome D: CUT implements wrong convention (does not externally conform with the reference).
	Outcome E: Fault in the reference compiler's caller. Fault in the CUT's callee.		Outcome F: Fault in the CUT's callee.
	Outcome G: Fault in the reference compiler's caller.		Outcome H: Not a possible outcome.
	Outcome I: Fault in reference compiler's callee. Fault in CUT's caller.		Outcome J: Fault in the CUT's caller.
	Outcome K: Fault in reference compiler's callee.		Outcome L: Not a possible outcome.
	Outcome M: Two conventions. One shared between the reference compiler's callee and CUT's caller, and *vice versa*.		Outcome N: Not a possible outcome.
	Outcome O: Not a possible outcome.		Outcome P: No faults detected.

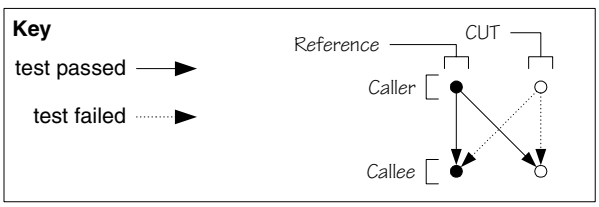

Table IV. All outcome configurations.

In testing the compilers, we checked for two types of conformance: internal and external. Compiler A internally conforms if code that it generates for a caller can properly call code for a callee that it generated. We denote this using $A \xrightarrow{c} A$. Compiler A externally conforms if its caller code can call another compiler B's callee code, and *vice versa* ($A \xrightarrow{c} B$ and $B \xrightarrow{c} A$). Thus, the callees and callers are compiled using each of the compilers under test. This results in n object versions for n compilers. Each caller version is then linked with the callee that was generated by the same compiler. This results in the n tests necessary to verify internal conformance for this test case. To establish external conformance, we could naively link each caller to each callee, which would yield $2n^2$ tests. However, we can do better. Recognizing that procedure call ($\xrightarrow{c}$) is symmetric we can easily reduce this to n^2 (since if $A \xrightarrow{c} B$, then $B \xrightarrow{c} A$). Furthermore, procedure call is also transitive, so if $A \xrightarrow{c} B$ and $B \xrightarrow{c} C$, then $A \xrightarrow{c} C$. This reduces the number to $2n - n$ as pictured in Figure 9. Each compiler's caller is linked to the reference compiler's callee. This facilitates the isolation of which compiler does not conform when an error is detected.

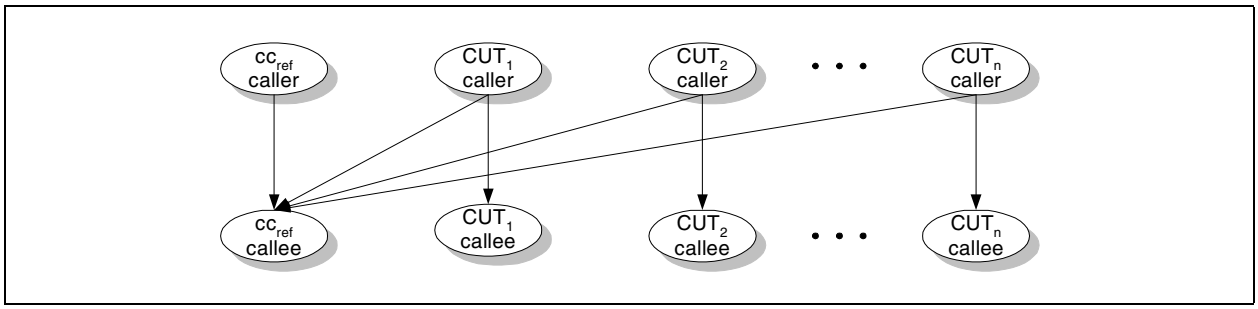

Figure 9. Determining conformance of n compilers.

The results of running both internal and external tests on the compiler set for the MIPS are shown in Table V. We found both internal and external conformance errors in all of the tested compilers. Table V reports internal and external errors separately. Within each class, the number of actual tests that failed and the number of faults that caused failure are indicated[1]. The numbers reported in the fault columns indicate the approximate number of actual coding errors resulting in test failures.

1. These numbers include tests of both standard procedure calls and variadic procedure calls.

These numbers are only approximate. We tried, as best we could, to glean this information from the results of tests. More accurate numbers can only be obtained by examining the compiler's source.

	Internal		External	
Compiler	**Failed Tests**	**Faults**	**Failed Tests**	**Faults**
cc (native)	2,346	1	2,346	1
gcc (1.38)	2,370	2	2,567	3
gcc (2.1)	0	0	2,346	1
gcc (2.4.5)	1	1	2,374	3
lcc (1.9)[a]	0	0	0	0
lcc (3.3)	2,407	2	2,407	2
vpcc/vpo	2,346	1	486	3
Total	9,470	7	12,526	13

Table V. Results of running the MIPS test suite on several compilers.

a. Version 1.9 of lcc was not tested using *varargs* because we could not get the compiler to accept *varargs* callees. This could either be a problem with the compiler, or the particular version of `stdarg.h` on our machine.

4.4.1 Standard Procedure Calls

Internal conformance errors were found in two versions of *gcc*. *gcc* 1.38 failed 24 tests that focus on passing structures in registers. Structures between nine and 12 bytes in size (three words) are not properly passed starting in the second argument register. Procedure signatures that correspond to these tests include:

```
void (int, struct(9-12));
```

gcc 2.4.5 fails a single test. The fault occurs with procedures with the signature:

```
void (struct(1), struct(1), struct(1));
```

gcc 2.4.5 fails to even compile a procedure with this signature[1]. The fact that *gcc* 2.1 does not have this error indicates that the error was *introduced* after version 2.1. This supports our conjecture that such

1. The error returned by *gcc* 2.4.5 was:
   `gcc: Internal compiler error: program cc1 got fatal signal 4.`

method of automatic testing is extremely useful throughout the development and maintenance life-cycle of a compiler.

External conformance errors were more prevalent. *gcc* 1.38 does not properly pass 1-byte structures in registers. This results in 208 test case failures. *gcc* 1.38 and 2.4.5 cannot pass a structure in the third argument register when that structure is followed by another. The fault occurs with signatures matching:

```
void (int, int, struct(1-4), struct(any));
```

This results in another 13 test failures. Finally, *vpcc/vpo* has 486 tests that fail. Two faults are responsible: 1) structures are not passed properly in registers, and 2) 1 to 4-byte structures are not passed in memory correctly if they are immediately followed by another structure. These match signatures:

```
void (int, int, int, int, struct(1-4), struct);
```

4.4.2 Variadic Procedure Calls

Procedures that take variable-length argument lists (variadic functions) are written using one the of two standard header files: `varargs.h` (for traditional **C**) and `stdarg.h` (for ANSI **C**). These files provide a standard interface for the programmer to write variadic functions. Because a variadic function's caller uses the standard procedure calling convention, the variadic callee must also conform to this convention. The following paragraphs detail the results of calling callees that are implemented using *varargs*/*stdarg*.

Most variadic functions in **C** have signatures similar to the standard library function `printf`:

```
void func(char *, ...);
```

The function determines the number of arguments from the first parameter. However, functions of the form:

```
void func(double, ...);
```

are also valid. When running test cases that contained variadic functions whose first argument was a `double`, we found that none of the compilers, including the reference compiler, properly implemented the calling convention. The source of this difficulty is that until the type of the argument is known, the callee cannot determine whether to fetch the first argument from the floating-point regis-

ter or the integer register. Most implementations of *varargs* dump the contents of the argument-passing registers to the stack in the function's prologue. For calling conventions like the MIPS, a more sophisticated solution must be used. This error caused 2,346 test cases to fail for all of the compilers. Version 2 releases of *gcc* managed to avoid this problem at the expense of interoperability; their generated callees do not conform to the established calling convention.

From these results, obviously the state-of-the-art in compiler testing is inadequate. Because these are production-quality compilers, each of them has undoubtedly undergone rigorous testing. However, hand development of test suites is an arduous and itself error-prone task. Furthermore, because these tests are target specific, they must be revisited with each retargeting of the compiler. In contrast, by using automatic test generators that are target sensitive, compilers can quickly be validated before each release.

5 Conclusions

Building compilers that generate correct code continues to be a difficult problem. Current implementations of calling sequence generators often contain errors. This comes from the lack of a formal model and implementation mechanism that can guarantee completeness and consistency properties. We have presented such a formal model, called P-FSA's, for procedure calling conventions that can ensure these properties. A P-FSA that models a convention can be automatically constructed from the convention's specification. During construction, the convention can be analyzed to determine if it is complete and consistent. The resulting P-FSA can then be directly used as an implementation of the convention in an application.

Although it is possible to automatically generate the calling sequence generator using P-FSA's, some work is required to retrofit an existing compilation system to use them. Fortunately, it is possible to reap the benefits of P-FSA without any modification of the compiler. Using automated compiler tools and testing, one can significantly increase the robustness of *any* compiler. We have combined these two techniques, in a new way, that further closes the gap between actual compiler implementations and the ever-sought-after correct compiler. By using a formal model of procedure calling conventions, we have designed and implemented a technique that automatically identifies boundary test

cases for calling sequence generators and diagnoses the nature of the fault. We then applied this technique to measure the conformance of a number of production-quality compilers for the MIPS. This system identified a total of a least 20 faults in the tested compilers. These errors were significant enough to cause over 2,300 different test cases to fail. Clearly, this technique is effective at exposing and isolating faults in calling sequence generators of mature compilers. Undoubtedly, it would be even more effective during the initial development of a compilation system.

6 References

[1] M. W. Bailey and J. W. Davidson, "A formal model and specification language for procedure calling conventions," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 298–310, January 1995.

[2] S. C. Johnson and D. M. Ritchie, "The C language calling sequence." Bell Labs.

[3] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice-Hall, second ed., 1988.

[4] W. L. Johnson, J. H. Porter, S. I. Ackley, and D. T. Ross, "Automatic generation of efficient lexical processors using finite state techniques," *Communications of the ACM*, vol. 11, no. 12, pp. 805–813, 1968.

[5] T. A. Proebsting and C. W. Fraser, "Detecting pipeline structural hazards quickly," in *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 280–286, 1994.

[6] T. Müller, "Employing finite automata for resource scheduling," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 12–20, 1993.

[7] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 35, no. 5, pp. 1045–1079, 1955.

[8] F. C. Hennie, "Fault detecting experiments for sequential circuits," in *Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design*, pp. 95–110, November 1964.

[9] Z. Kohavi, *Switching and Finite Automata Theory*. McGraw-Hill, second ed., 1978.

[10] D. P. Sidhu and T.-K. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Transactions of Software Engineering*, vol. 15, pp. 413–426, April 1989.

[11] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, pp. 591–603, June 1991.

[12] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours," *IEEE Transactions on Communications*, vol. 39, pp. 1604–1615, November 1991.

[13] M. Yannakakis and D. Lee, "Testing finite state machines: Fault detection," *Journal of Computer and System Sciences*, vol. 50, pp. 209–227, 1995.

[14] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture validation for processors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 404–413, 1995.

[15] C. W. Fraser and D. R. Hanson, "A code generation interface for ANSI C," *Software–Practice and Experience*, vol. 21, no. 9, 1991.

[16] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings, 1995.

[17] R. M. Stallman, *Using and Porting GNU CC (Version 2.0)*. Free Software Foundation, Inc., February 1992.

[18] M. E. Benitez and J. W. Davidson, "A portable global optimizer and linker," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 329–338, July 1988.

[19] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[20] A. W. Appel. Personal Communication, May 1996.