Reprinted from *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

Target-Sensitive Construction of Diagnostic Programs for Procedure Calling Sequence Generators[†]

Mark W. Bailey
Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
USA

Abstract

Building compilers that generate correct code is difficult. In this paper we present a compiler testing technique that closes the gap between actual 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 target-specific test suites, our automated testing tool has exposed bugs in every compiler tested. These compilers include ones that have been in heavy use for many years. The detected bugs cause more than 14,000 test cases to fail.

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* that implement procedure calls. We have developed a language, called CCL, for specifying procedure calling conventions. CCL specifications are used to automatically generate calling sequences for a retargetable optimizing compiler [BD95]. In doing so, we realized that CCL 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 specifications 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.

One 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* [Joh], must be generated by the code generator. This aspect of the code generator, which we refer to as 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.

As part of research whose objective is to develop more retargetable optimizing compilers, we have developed a formal specification language for describing procedure calling conventions. This language, called CCL, has been used to generate automatically the calling sequence generator for a compiler [BD95]. The compiler, called *vpcc/vpo*, is a retargetable optimizing compiler for the **C** language that has been targeted to over a dozen different architectures [BD88, BD94].

The procedure calling convention for a target machine is described using CCL. The resulting specification is processed by a CCL interpreter. The interpreter can generate tables that can be used in the calling-convention-specific portion of *vpcc/vpo*, or in a test suite generator. This process is pictured in Figure 1a. 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.

As shown in Figure 1b, a calling convention description can be used to automatically generate test cases that exercise a compiler's calling sequence generator. A test case is composed of two procedures, each in its own file. One file is compiled by the compiler under test, 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 objects 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.

This paper makes several contributions. First, a method for automatically testing implementations of procedure calling conventions is presented. Using this technique we have found bugs in mature **C** compilers. This technique, which uses a formal specification of procedure calling conventions, methodically generates tests that offer complete coverage of the problem domain. Second, an algorithm for intelligently selecting important tests from the complete coverage suite is introduced. These tests include bound-

[†] This work was supported in part by National Science Foundation grant CCR-9214904. Authors' addresses: `mark@virginia.edu`, `jwd@virginia.edu`.

PLDI '96 5/96 PA, USA
© 1996 ACM 0-89791-795-2/96/0005....$3.50

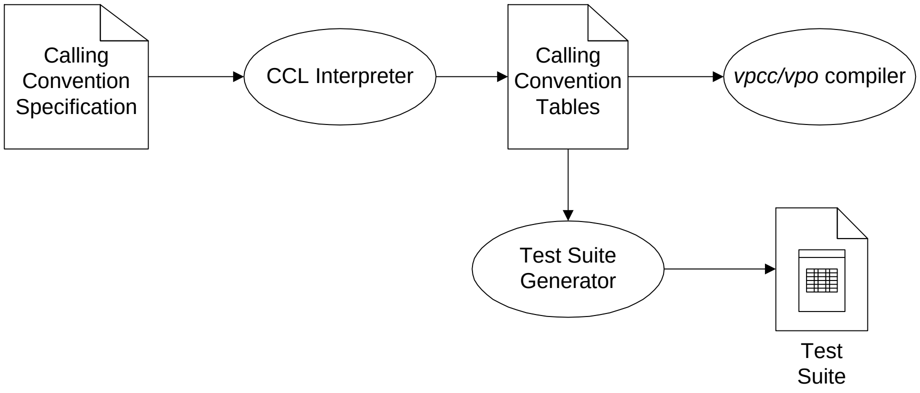

Figure 1a: How CCL specifications are used.

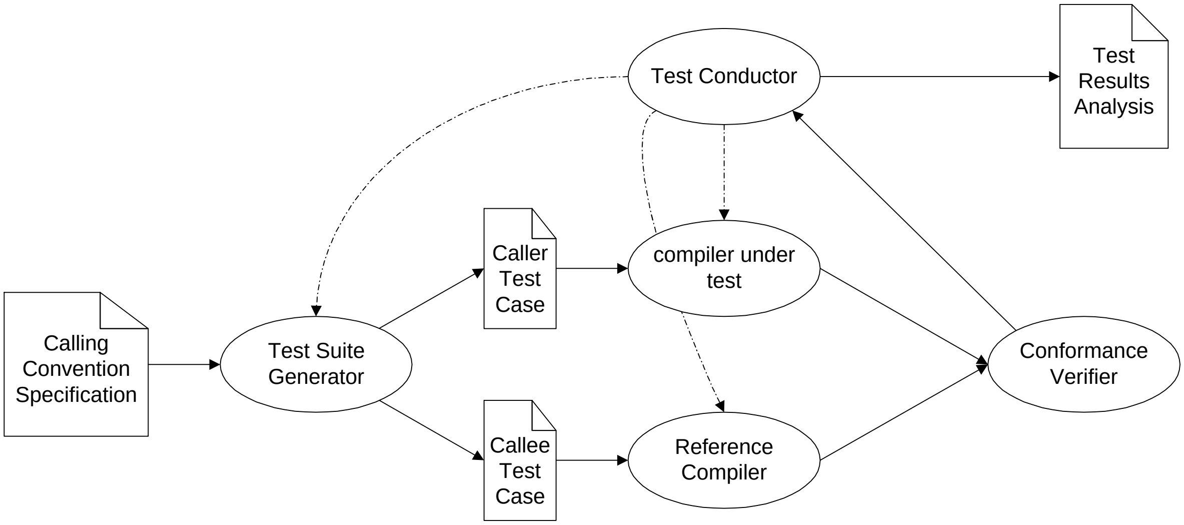

Figure 1b: Using test suites to determine convention conformance of a compiler.

ary 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 can isolate errors more effectively than tests from a general test suite. Finally, a method for quickly determining the conformance of multiple compilers at once is described.

2 Procedure Calling Conventions

A calling convention is the set of rules to which a caller and callee must conform. Figure 2 contains the calling convention rules for a hypothetical machine. Consider the following ANSI **C** prototype for a function **foo**:

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

For the purpose of transmitting procedure arguments for our simple convention, we are only interested in 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, which for the above function prototype is:

```
int foo(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 $\mathbf{a}^1$, $\mathbf{a}^2$, $\mathbf{a}^3$, and $\mathbf{a}^4$ are 32-bit argument-transmitting registers.
2. Arguments are also passed on the stack in increasing memory locations starting at the stack pointer (**M[sp]**).
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 $\mathbf{a}^1$ and $\mathbf{a}^2$.
9. Values of registers $\mathbf{a}^6$, $\mathbf{a}^7$, $\mathbf{a}^8$, and $\mathbf{a}^9$ must be preserved across a procedure call.

Figure 2: Rules for a simple calling convention.

With **foo**'s signature, we can apply the calling convention in Figure 2 to determine how to call **foo**. Arguments to **foo** would be placed in the following locations:

- **p1** in register $\mathbf{a}^1$,
- **p2** in register $\mathbf{a}^2$,

- **p3** in register $\mathbf{a^3}$, and
- **p4** on the stack in **M[sp:sp + 7]** (**M** denotes memory).

Notice that although register $\mathbf{a^4}$ is available, **p4** is placed on the stack since it cannot be placed completely in the remaining register (rule 4). Such restrictions are common in actual calling conventions.

Now that we have seen how arguments are transmitted for a simple example, we can describe the objects in our model. The primary objects of interest are machine resources. A machine resource is simply any location that can store a value. Examples include registers and memory locations, such as the stack. Defining where required values are located is accomplished by specifying a mapping from one resource to another. We call such a mapping a *placement*.

CCL descriptions define placement functions for both procedure arguments and return values. They also describe other aspects of calling conventions such as frame layout and stack allocation. However, these features of CCL are not considered in this paper since the focus of our test suite generator is verifying the implementation of placement functions.

3 The Formal Model

We use finite state automata to model each placement in a calling convention. 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 placement is written to the output.

Placement functions are described in terms of finite resources, infinite resources, and selection criteria. A set of finite resources $R = \{r_1, r_2, \ldots, r_n\}$ are used to represent machine registers, while an infinite resource $I = \{i_1, i_2, \ldots\}$ is used represented the stack. The *selection criteria* $C = \{c_1, c_2, \ldots, c_m\}$ 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 $w \in (C^*, C^*)$. 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 $\|d\| = 3$. So, each state is labeled by a string from the language $\{0, 1\}^4\{0, 1\}^3$. The output of M 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 [Mea55]. We define a P-FSA, M, as a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, 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 placement strings,
- the transition function $\delta{:}Q \times \Sigma \to Q$,
- the output function $\lambda{:}Q \times \Sigma \to \Delta^+$, and
- q_0 is the state labeled by $0^n w$ where $\|w\| = \|d\|$, and w is the initial state of d.

The P-FSA that corresponds to our hypothetical calling convention is shown in Figure 3. Its output function λ is shown in Table I. The states of the automaton represent that state of allocation for the machine resources. For example, the state labeled q_2 (**1100 000**) represents the fact that register $\mathbf{a^1}$ and $\mathbf{a^2}$ have been allocated, but that $\mathbf{a^3}$, $\mathbf{a^4}$ and stack locations have not been allocated. The transitions between states represent 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**.

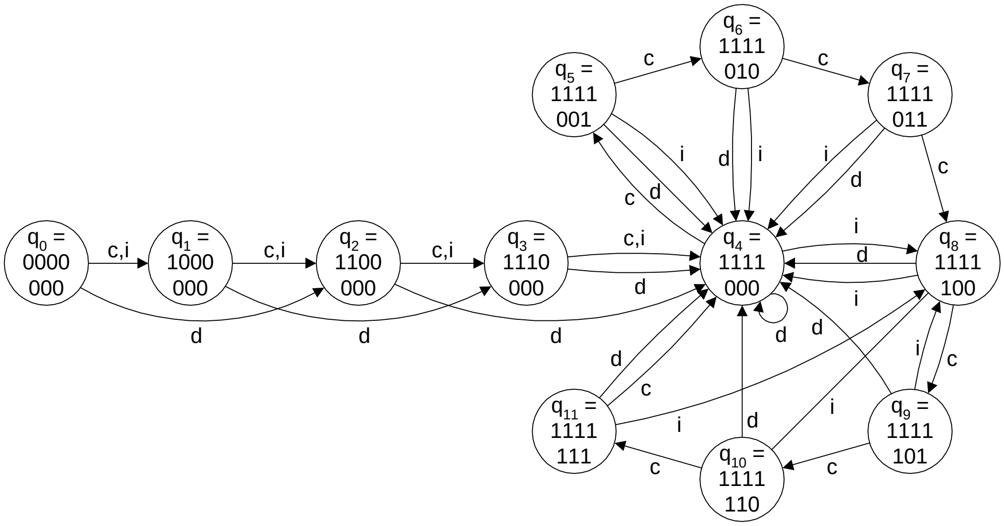

Figure 3: A simple placement finite-state automaton.

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	a^1	a^2	a^3	a^4	000	001	010	011	100	101	110	111
int	a^1	a^2	a^3	a^4	mem_1[a]	mem_2[b]	mem_2	mem_2	mem_2	mem_1	mem_1	mem_1
double	a^1a^2	a^2a^3	a^3a^4	mem_3[c]	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3	mem_3

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

[a] mem_1 = 000 001 010 011
[b] mem_2 = 100 101 110 111
[c] mem_3 = 000 001 010 011 100 101 110 111

The signature:

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

will take the P-FSA in Figure 3 from state q_0 to q_4 producing the string (**a**$^{\mathbf{1}}$ **a**$^{\mathbf{2}}$) (**a**$^{\mathbf{3}}$ **a**$^{\mathbf{4}}$) (**000**) (**100 101 110 111**) along the way. From the string, we can derive the placement of the **phred**'s arguments. The first **double** is placed in registers **a**$^{\mathbf{1}}$ and **a**$^{\mathbf{2}}$, the second in registers **a**$^{\mathbf{3}}$ and **a**$^{\mathbf{4}}$, the **char** at the first stack location and the **int** starting in the fifth stack location.

4 Test Vector Selection

In order to test a compiler's implementation of a calling convention, we must select a set of programs to compile. 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. Obviously, one cannot test all signatures since the set of signatures, $S = \{(C^*, C^*)\}$, is infinite. 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 [Hen64, Koh78]. There are numerous approaches to this problem, most of them 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, or 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 a small number of states, and a small input alphabet, this may be acceptable. However, the number of such paths for an FSA is $O(\|\Sigma\|^{\|Q\|})$. 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 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.

Another, simpler, approach is to guarantee that each transition is exercised at least once. Since there are no more than $\|Q\|\|\Sigma\|$ 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 3, the two signatures:

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

cover all **int** and **double** transitions leaving states $q_{0\text{-}3}$. 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 4a. An erroneous implementation, shown in Figure 4b, contains an extra state q_1' that is reached on initial input **b**. The two strings, **aaa** and **bbb** completely cover the 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.

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

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.

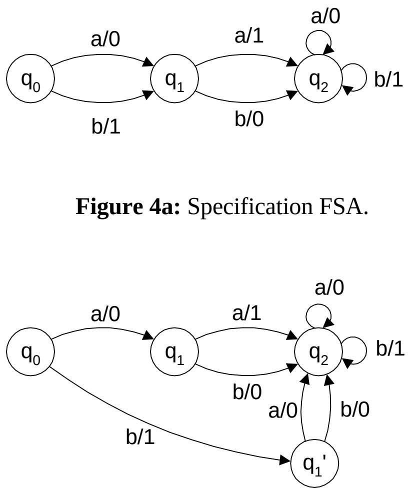

Figure 4a: Specification FSA.

Figure 4b: Implementation FSA.

Figure 4: Example FSA where a fault will not be detected

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 specification FSA. As shown in Figure 5, 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 4. Furthermore, it 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 3, 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_3, **double**), (q_4, **double**)). Such a set would not need to include **int int double double** to cover the same transition pair.

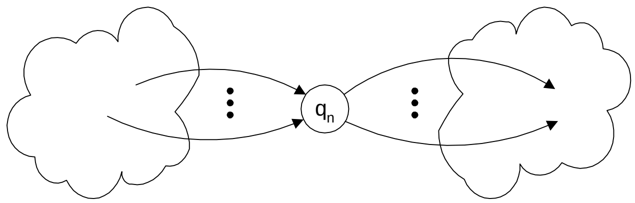

Figure 5: Entering and exiting transitions for a state.

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 confidence level.

An algorithm for generating transition-pair paths is shown in Figure 8 (in the appendix). 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) are 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.

Machine	*Transition Paths*	*Transition-pair paths*	*Acyclic paths*
DEC VAX	3	12	3
M68020 (Sun)	24	324	96
SPARC (Sun)	224	7434	> 10^8
M88100 (Motorola)	720	22,412	> 10^8
MIPS R3000 (DEC)	772	5655	$8\text{x}10^8$

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

5 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 broken into 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 composed 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 7 (in the appendix) shows an example test case for the **C** signature **void (int, double, struct(2)[2])**. The caller (Figure 7a) loads each of the arguments 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 $0 < i \le m$, there exists no index $0 < j \le m$ and $j \ne i$ such that $B[j + k] = B[i + k]$ for all $0 \le k < 2$. 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 easy to identify if an argument has been shifted or misplaced since each argument’s value is guaranteed to be unique. The callee (Figure 7b) receives the values, and checks them against the expected values. If the values do not match, an error condition is signalled.

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 can be tricky. For each test case that we generate we also generate a *varargs* version to verify that this standard convention is implemented correctly. However, we do not include an example here.

[2] We denote a structure whose size is n bytes as struct(n).

6 Results

We used our technique for selecting test vectors to test a number of 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, a version of Fraser and Hanson's *lcc* [FH91] compiler, several versions of GNU's *gcc* [Sta92], and a previous version of our own **C** compiler that used a hand-coded calling sequence generator. 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.

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 6. If a reference compiler (an operational definition of the calling convention), C_{ref}, is used, Figure 6 would look different. Each compiler's caller/callee would be linked to the reference compiler's callee/caller. This facilitates the isolation of which compiler does not conform when an error is detected.

The results of running both internal and external tests on the compiler set for the MIPS are shown in Table IV. We found both internal and external conformance errors in all of the tested compilers. Table IV reports internal and external errors separately. Within each class, the number of actual tests that failed and the number of faults that caused test failure are indicated[3]. The numbers reported in the fault columns indicate the approximate number of actual coding errors resulting in test failures. 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.

6.1 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 9 and 12 bytes in size (3 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[4]. 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 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);
```

6.2 Variadic Procedure Calls

Procedures that take variable-length argument lists (variadic functions) are written using one 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 perfectly 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 difficulty stems from the fact that until the type of the argument is known, the callee cannot determine whether to fetch the first argument from the floating-point register 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 2346 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, it is clear that 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

[3] These numbers include tests of both standard procedure calls and variadic procedure calls.

[4] The error returned by *gcc* 2.4.5 is: `gcc: Internal compiler error: program cc1 got fatal signal 4`.

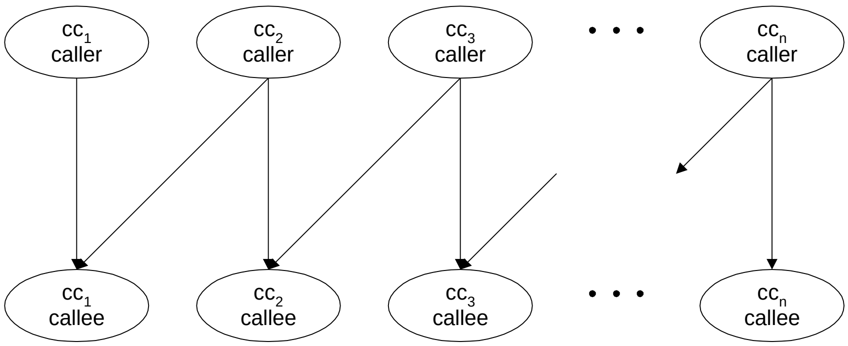

Figure 6: Determining conformance of *n* compilers.

	Internal		*External*	
Compiler	***Failed tests***	***Faults***	***Failed Tests***	***Faults***
cc (native)	2346	1	2346	1
gcc (1.38)	2370	2	2567	3
gcc (2.1)	0	0	2346	1
gcc (2.4.5)	1	1	2374	3
lcc (1.9)[a]	0	0	0	0
lcc (3.3)	2407	2	2407	2
vpcc/vpo	2346	1	486	3
Total	14162	9	14872	13

Table IV: Results of running 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.

sensitive, compilers can be quickly be validated before each release.

7 Related Work

The automatic generation of test suites has received much attention recently in the area of conformance testing of network protocols [SL89]. The purpose of the suite is to determine if the implementation of a communication protocol adheres to the protocol's specification. In many cases, the protocol specification is provided in the form of a finite-state machine. This has resulted in many methods of test selection including the Transition tour, Partial W-method [FBK+91], Distinguishing Sequence Method [Koh78], and Unique-Input-Output method [ADLU91]. These methods are derivatives of the checking experiment problem where an implementation is checked against a specification FSM [YL95]. What distinguishes these methods from ours is that a bound on the number of states in the implementation FSM is assumed. Because we have no practical bound on the number of states in the implementation, their work is not applicable. Finally, a similarly related field is the automatic verification of digital circuits [Hen64, HYHD95].

8 Summary

Building compilers that generate correct code continues to be a difficult problem. Using automated compiler tools and testing, one can significantly increase the robustness of a 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 formal specifications of procedure calling conventions, we have designed and implemented a technique that automatically identifies boundary test cases for calling sequence generators. 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 22 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 bugs in calling sequence generators of mature compilers. It would be even more effective during the initial development of a compilation system.

References

[ADLU91] Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604—1615, November, 1991.

[BD88] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the ACM SIGPLAN ' 88 Conference on Programming Language Design and Implementation*, pages 329—338, July 1988.

[BD94] Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. In *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, pages 105—124, March 1994.

[BD95] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the 22nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298—310, January 1995.

[FBK+91] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591—603, June 1991.

[FH91] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI-C. *Software—Practice and Experience*, 21(9), 1991.

[Hen64] F. C. Hennie. Fault detecting experiments for sequential circuits. In *Proceedings of the Fifth Annual Sympo-*

sium on Switching Theory and Logical Design, pages 95—110, November 1964.

[HYHD95] Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architecture validation for processors. In *ISCA95*, pages 404—413, 1995.

[Joh] S. C. Johnson. A tour through the portable c compiler.

[Koh78] Zvi Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, second edition, 1978.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 35(5):1045—1079, 1955.

[SL89] Deepinder P. Sidhu and Ting-Kau Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions of Software Engineering*, 15(4):413—426, April 1989.

[Sta92] Richard M. Stallman. *Using and Porting GNU CC (Version 2.0).* Free Software Foundation, Inc., February 1992.

[YL95] Mihalis Yannakakis and David Lee. Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50:209—227, 1995.

Appendix

```
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(0x44026097l,
                         double_arg_2,
                         struct_2_arg_3);
}
```

Figure 7a: 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 != 0x44026097l) {
        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 7b: Code generated for callee.

Figure 7: Example test case.

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, $\emptyset$, 0);

Algorithm:

```
function TRANSITION-PAIRS($q$, $w$, $V$, cycle)
    paths $\leftarrow \emptyset$;
    for each $a$ where $a \in \Sigma \wedge \delta(q, a)$ is defined do        // For each transition from state q...
        if cycle $\neq 1 \wedge (q, a) \notin T$ then                            // No cycles and (q, a) is new
            if $q \notin V$ then                                                 // If there is no cycle
                $T \leftarrow T \cup \{(q, a)\}$;                                // Mark transition as followed
                cycle $\leftarrow$ 0;                                            // Indicate no cycle
            else
                cycle $\leftarrow$ 1;                                            // Indicate cycle
            end if
            $P \leftarrow$ TRANSITION-PAIRS($\delta(q, a)$, $wa$, $V \cup \{q\}$, cycle);   // Compute paths from here
            paths $\leftarrow$ paths $\cup\ P$;
        end if
        paths $\leftarrow$ paths $\cup \{wa\}$;                                  // Add this path to paths
    end for
    return paths;                                                                // Return paths from q
end function
```

Figure 8: Test vector generation algorithm.