Submitted to *ACM Transactions on Programming Languages and Systems.*

Construction of Systems Software Using Specifications of Procedure Calling Conventions

MARK W. BAILEY and JACK W. DAVIDSON
University of Virginia

Applications, such as compilers and debuggers, which generate, or process procedures at the machine-language abstraction level require knowledge of the procedure calling convention. Currently, applications that process procedures implement conventions in an *ad-hoc* manner. The resulting code is complicated with details, difficult to maintain, and often plagued with errors. In this paper, we describe the only known formal model and specification language for procedure calling conventions. The model and language, in combination, facilitate the accurate specification of conventions that can be shown to be both consistent and complete. Further, we show how the convention specifications can be used to automatically generate that part of the code generator responsible for generating procedure calls. Finally, we discuss a new compiler testing technique that uses the specifications to further close the gap between actual compiler implementations and correct compilers. The technique, which uses a target-sensitive test suite generator, has exposed and diagnosed faults in several **C** compilers.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**] Processors—*code generation; compilers*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions and subroutine*s; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids; test data generators*

General terms: Testing, Code Generation, Procedure Call, Specification

Additional Key Words and Phrases: Procedure Calling Convention, Test Suite Generation, Program Fault Diagnosis

1. INTRODUCTION

The procedure calling convention impacts the operation of many system software components. The interface between procedures, which is established by the calling convention, facilitates separate compilation of program modules and interoperability of programming languages. The *procedure calling convention* 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*). What makes calling conventions unique and interesting is that they are not implementation dependent or entirely language dependent. Instead, 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. Further, code that implements the calling convention must be generated by the compiler and understood by other systems software.

1.1 Motivation

Currently, information about a particular calling convention can be found by: looking in the programmer's reference manual for the given machine, or reverse-engineering the code generated by one of its compilers. Reverse-engineering a compiler has many obvious shortcomings. Using the programmer's reference manual may be equally problematical. As with much of the information in the programmer's manual, the description is likely to be written in English and is liable to be ambiguous, or inaccurate. For example, in the MIPS programmer's manual [Kane and Heinrich 1992] the English description is so difficult to understand that the authors provide fifteen examples, several of which are contradictory [Fraser 1993]—and this is the *second* edition of the programmer's manual. Furthermore, the convention, once understood, is difficult to implement. For example, the GNU ANSI **C** compiler fails on an example listed in the manual. Digital Equipment Corporation, in recognizing the problem, has published a calling standard document for their Alpha series processors [Digital Equipment Corporation 1993] that exceeds 100 pages[1]. Thus, it should be clear that there is a need for accurate, concise descriptions of procedure calling conventions and software to use them.

1.2 Applications

Any application that must process, or generate procedures at the machine-language abstraction level is likely to need to know about a procedure calling convention. Example applications include compilers, debuggers, linkers and evaluation tools such as profilers. The code that implements the calling convention in these applications lends itself to automatic generation. Often, the convention itself is not difficult to understand or implement, for a given instance of a procedure call. However, a general solution that covers all possible cases is difficult to implement correctly.

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* (Calling Convention Language), has been used to generate automatically the calling sequence generator for a compiler [Bailey and Davidson 1995]. The compiler, called *vpcc/vpo*, is a retargetable optimizing compiler for the **C** language that has been targeted to over a dozen different architectures [Benitez and Davidson 1988; Benitez and Davidson 1994].

The procedure calling convention for a target machine is described using CCL. The resulting specification is processed by an interpreter that can generate tables that can be used in the calling-convention-specific portion of *vpcc/vpo*, or in a test suite generator. Fig. 1 shows this process. The test suite generator uses information from the table to build a test suite for the specific calling convention. The test suite can be used to either confirm that the *vpcc/vpo* implementation properly uses the convention tables, or confirm that another, independent, compiler conforms to the convention described in the CCL specification.

---

[1]Although this document also includes information on exception handling and information pertinent to multithreaded execution environments, more than 42 pages are devoted to documenting the calling convention.

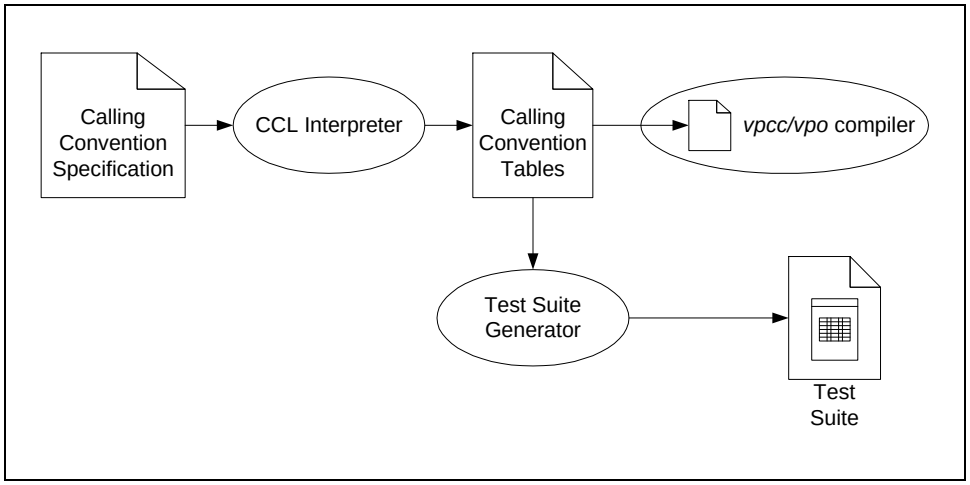

Fig. 1. How CCL specifications are used.

2. PROCEDURE CALLING CONVENTIONS

To facilitate local compilation of procedures, compiler developers establish rules about how procedures interact. These rules establish an agreement between the caller and callee on how information and control are passed between the two, as well as how and who will maintain the state of the machine. Collectively, these rules are known as the procedure calling convention.

2.1 A Simple Calling Convention

To aid in our discussion of calling conventions, we use a simplified example calling convention. Fig. 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.

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

— p1 in register a^1,

— p2 in register a^2,

— p3 in register a^3, and

— p4 on the stack in M[sp:sp + 7] (M denotes memory).

(1) Registers a^1, a^2, a^3, and 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 a^1 and a^2.
(9) Values of registers a^6, a^7, a^8, and a^9 must be preserved across a procedure call.

Fig. 2. Rules for a simple calling convention.

Notice that although register a^4 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, Language and Implementation

The first thing to notice about our simple calling convention is the lack of detail. There are many questions that are left unanswered. Among them are:

(1) What order are the procedure's arguments evaluated?
(2) What order are the procedure's arguments placed in registers and on the stack?
(3) Where are the persistent registers stored?
(4) Which persistent registers need to be saved?
(5) What is the activation frame layout?

Each of these questions must be answered in order to produce a working implementation. These questions are answered by two other elements that interact with the procedure calling convention: the definition of the procedure's source language and the language's implementation. In this work, we have made a conscious effort to separate the concepts of calling convention, language definition and implementation.

The choice to isolate the concepts of the convention from those of the language definition is an obvious one. To facilitate inter-language procedure calls, a single convention separate from the language definition, must be available. There are, however, features of the source language that may be present in the convention. For example, in our hypothetical convention, *where* an argument is placed is determined, in part, by the type of the argument. Such language features cannot be avoided in the description of the convention, but they should be kept to a minimum. Also, it illustrates what features both languages must share to make inter-language procedure calls possible at all.

The need for the second separation, between the convention and the language implementation, may be less obvious. Compiler writers commonly refer to the mechanism by which procedure calls are made as either the calling convention, or the calling sequence. Although these two terms are frequently used interchangeably, they are separate concepts and we treat them as such. Without additional information, the calling convention itself does not provide enough information to produce an implementation. The calling sequence, on the other hand, is an implementation of the calling convention. It is a sequence of machine instructions that implement a procedure call. There may be many calling sequences for a given calling convention. Furthermore, since the sequence implements the convention, it is impossible for the caller to determine if the callee is using the same sequence, and vice versa. Thus, while it is imperative that a caller and a callee use the same calling convention, it is not necessary that they use the same calling sequence.

2.3 Separating Convention from Sequence

An important result of this work is the identification of calling convention and calling sequence as separate concepts. Although at first this distinction may seem unnatural, it has many benefits. The reason it seems unnatural is that the two concepts are so closely coupled. It is impossible to discuss calling sequences without calling conventions. However, the reverse is not true. By extracting the concept of convention from the calling sequence, we are able to more accurately model the interaction between procedures and the interaction between system software that process procedures.

When discussing calling conventions, we have found it useful to have a litmus test that helps us identify what features of the procedure call are part of the calling convention, and what features are part of the calling sequence. We ask the following question:

> *If I change the implementation of this feature on one side of the procedure call, will it impact the other side of the call?*

If the answer to this question is *yes*, then the feature is part of the calling convention. If *no*, the feature is part of the calling sequence. For example, if the callee changes where it stores the values of persistent registers that it uses, the caller need not be changed. Thus, where these values are stored is a feature of the calling sequence. Conversely, if the callee changes where it stores its return value, the caller must also be changed so it can properly retrieve the value upon return. Therefore, the placement of the return value is a matter of calling convention.

2.4 Interfaces and Agents

So far, we have referred to the procedure call interface. Actually, there are two interfaces: the procedure call interface and the procedure return interface as shown in Fig. 3. We model the actions and responsibilities on each side of these interfaces using agents. An *agent* ensures that its side of the interface satisfies the requirements of the calling convention. These agents are the *whom* in the definition of the calling convention. For the procedure call interface, there are the *caller prologue* and *callee prologue* agents that are responsible for correctly passing the procedure arguments and con-

structing an environment that the callee can execute in. For the procedure return interface, there are the *callee epilogue* and *caller epilogue* that are responsible for correctly passing the procedure return values and restoring the environment of the caller. The responsibilities of each of the four agents are closely related. The caller prologue and callee prologue agents must agree on how to pass information, as must the caller epilogue and callee epilogue. Additionally, actions of the epilogue agents must be symmetric to the actions of the prologue agents to properly restore the environment (*e.g.*, if the call decrements the stack pointer, the return must increment it). It is precisely these restrictions that make it difficult correctly construct a calling sequence.

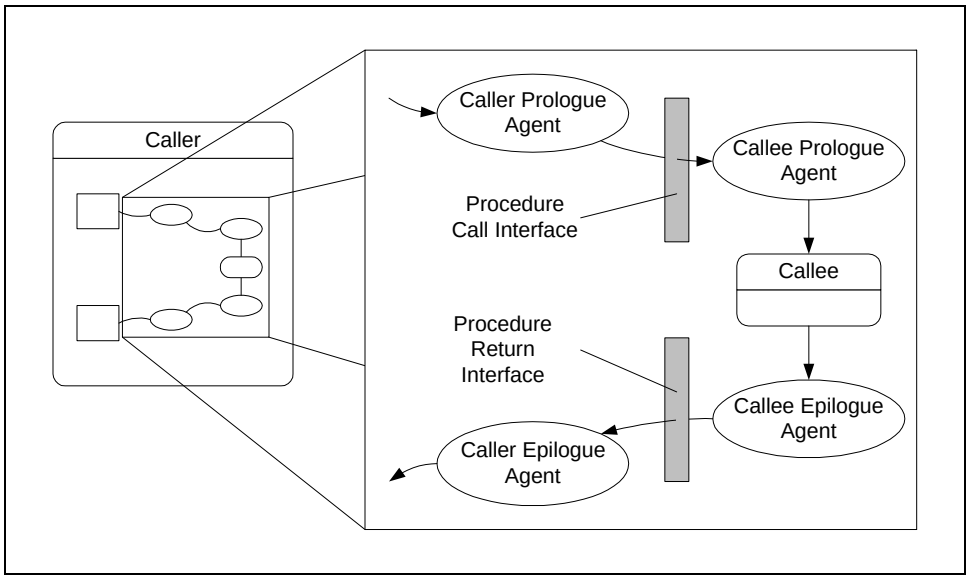

Fig. 3. The role of agents in procedure call and return interfaces.

2.5 Addressing

One responsibility of each agent is to maintain the environment in which procedures execute. Depending on the language and its implementation, the environment can contain arbitrary information. However, one aspect of the environment that almost all languages are likely to share is the concept of addressing. Addressing describes how a name in the source language is bound to a location in the implementation. For example, local variables are commonly found on the stack, while global variables may be referenced through a global space pointer.

Sometimes, to properly construct an environment for a procedure, the caller must provide to the callee details about the caller's environment. For example, in Pascal, where nested procedures can refer to variables in the scope of their containing procedures (up-level references), the caller must provide the callee with environment information for the callee to properly implement the scoping rules of the language. Using our litmus test, clearly the structure of the environment information is part of the call-

ing convention. If the structure changed, the callee would need to be changed so it could properly find variables that visible to the callee.

Although the structure of information that is transmitted between procedures is a matter of convention, we have not included it in our convention specifications. Just as it is reasonable to discuss calling convention rules using data types that are never formally defined, it is reasonable to specify how information is passed between procedures without defining its structure. We believe that description of the structure of information is itself an interesting and difficult problem that is best left as a future research effort. When such a description is developed, incorporating it into our specifications should require little effort.

2.6 Activation Frame Layout

An important decision that must be made when implementing a procedure calling convention is the layout of the procedure activation frame. An activation frame is one of several implementation choices for storing the information specific to a particular activation of a procedure. A surprising result of studying calling conventions is that a complete specification of the calling convention is unlikely to determine the frame layout.

Information that is typically found in a procedure activation frame includes: the procedure's parameters, locations for storing local variables and temporaries, space for saving the values of persistent registers, and space for any other environment information. Where this information is found in the frame is determined, in part, by the convention and, in part, by the implementation. The convention fixes the location of the procedure's arguments, while it is up to the implementation to specify where local variables are stored. Thus, any implementation must make some decisions about frame layout. Section 5.2 discusses how this is done in our implementation.

3. THE CCL SPECIFICATION LANGUAGE

In this section, we briefly describe the specification language that we use to describe procedure calling conventions. Once a convention is specified in CCL, we avoid the pitfalls related to using the programmer's reference manual, or reverse-engineering the compiler. The following sections present the key features of CCL and enough syntax of the language to understand the examples. The extended ASCII syntax shown is just the form that we chose to use. Another syntax could be used if it incorporated the underlying concepts of CCL.

3.1 Design Philosophy

In designing CCL, there were a number of features that we wanted to be present. First and foremost, the language had to be processed automatically. Second, we wanted the descriptions to be natural. Hence, the elements of the language had to reflect objects common to calling conventions. Third, the design had to avoid over-specification of conventions. To achieve this, we tried to exploit the symmetry of the procedure call to eliminate redundancy in the descriptions. Finally, the feature that received the least

priority was the syntax of the language. The selection of what symbols to use for operators, for example, was only of secondary concern.

We describe the key features of CCL by presenting a simple CCL description. Fig. 4 contains the complete specification for our hypothetical calling convention. CCL uses an extended ASCII character set and typographical elements such as bold face, superscripts and special fonts. The specifications are edited using a desktop publishing system and are automatically processed from this form.

The primary objects of CCL 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*. In CCL, machine resources are modeled as arrays with attributes in CCL. In addition, the language also includes many familiar types of expressions. They include:

— Sets: {2:9} ≡ {2,3,4,5,6,7,8,9}

— Ordered sets: <2,8,3,9,4,10>, <0:∞>

— Labeled sets: {char: 1, short: 2, longword: 4, float: 4, double: 8}

— Arrays: $\mathbf{M}[14] \equiv \mathbf{M}^{14}$, $<\mathbf{M}[\mathbf{r}^{14}:\mathbf{r}^{14}+31]> \equiv <\mathbf{M}[\mathbf{r}^{14}(32)]>$

— Operators: mod, ∑, ∧, ∈, ⊥

— Keywords: **external**, **alias**, **caller prologue**, **resources**, **map**, **set**

— Comments: *This is a comment*

Sets and arrays are the most important types in CCL. In combination, they provide a natural way to discuss features of a calling convention. The range operator (:) is used to build sets. For example, we can build an ordered set of machine resources that represents the first 8 locations of the stack using the expression <M[sp:sp+7]> which is equivalent to <M[sp], M[sp+1], … M[sp+7]> where sp is an alias for the register containing the stack pointer (another equivalent expression is M[sp:sp(8)]).

3.2 Outer Environment

The CCL language is a part of CSDL, which is a larger description system we are developing at the University of Virginia [Bailey and Davidson 1996]. Although CCL is used to capture information about a calling convention, a CCL description does not contain all necessary information to produce a calling sequence. Indeed, CCL descriptions are not complete by themselves. CCL descriptions require information from the outer environment to complete the descriptions. Information about the machine and language, such as the size of registers, the base data types and local procedure information, such as the amount of space needed for temporary variables, and which registers are used, are provided by other components of the CSDL description system. Four variables that are always defined by the outer environment are the special resources ARG, RVAL, and the corresponding special resource sizes ARG_TOTAL and RVAL_TOTAL. Since these values are always defined, they are implicitly declared as external values. All other variables whose values are provided by the outer environment are declared using the external statement.

```
1   external NVSIZE, SPILL_SIZE, LOCALS_SIZE
2   persistent {a^6, a^7, a^8, a^9}
3   alias sp ≡ a^5
4   caller prologue
5       view change
6           ∀ offset ∈ {−∞:∞}
7               M[sp + offset] becomes M[sp + offset + ARG_SIZE]
8       end view change
9       data transfer (asymmetric)
10          alias mindex ≡ sp:∞
11          alias argregs ≡ a^1:4
12          resources {<argregs, M^mindex>}
13          internal ARG_SIZE ← ∑(<M[addr].size | addr ∈ <mindex> ∧
14              M[addr].assigned>)
15          class regs ← <<register> | register ∈ <argregs>>
16          class imem ← <<M[addr]> | addr ∈ <mindex> ∧ addr mod 4 = 0>
17          class dmem ← <<M[addr]> | addr ∈ <mindex> ∧ addr mod 8 = 0>
18          ∀ argument ∈ <ARG^1:ARG_TOTAL>
19              map argument → argument.type ⊥ {
20                  char:       <regs, M^mindex>,
21                  int:        <regs, imem>,
22                  double:     <regs, dmem>,
23              }
24      end data transfer
25  end caller prologue
26  callee prologue
27      view change
28          ∀ offset ∈ {-∞:∞}
29              M[sp + offset] becomes M[sp + offset + SPILL_SIZE +
30                  LOCALS_SIZE + NVSIZE]
31      end view change
32  end callee prologue
33  callee epilogue
34      data transfer (asymmetric)
35          resources {a^1:2}
36          map RVAL^1 → <<<a^1>>>
37      end data transfer
38  end callee epilogue
39  caller epilogue
40  end caller epilogue
```

Fig. 4. A CCL description of the calling convention of Fig. 2.

A CCL description is typically language dependent as well. This is, in part, because the language definition influences the calling convention. For example, the **C** language [Kernighan and Ritchie 1978] defines a slightly different calling convention

than its successor ANSI **C** [Kernighan and Ritchie 1988]. One difference is that **C** always promotes arguments of type float to type double, while ANSI **C** does not. These differences are part of the calling convention, and are, therefore, present in the resulting CCL descriptions. Although ANSI **C** is now the standard, all of the examples in this paper assume the traditional **C** language calling convention since our compiler, which uses the descriptions, implements traditional **C**. However, we also have ANSI **C** descriptions available. Finally, although we only present descriptions for **C**, we believe that the calling convention of any Algol-based language is amenable to description in CCL.

3.3 Placement of Parameters and Return Values

Here, we examine the placement of procedure arguments. Return values are specified in the same manner. For placement of arguments, we focus on the data transfer statement within the caller prologue section of the description (lines 9–24). We use the alias statement to introduce the name argregs as a name for the parameter passing registers and mindex as a set of stack addresses (a^5 is the stack pointer). Line 12 defines the set of possible destinations for data placement, which we call the resources. Lines 15–17 specify classes, each of which defines a subset of these resources, where placements may start. Since the convention has two different alignment restrictions for memory, based on argument type, there is a corresponding class for each restriction and a class for the argument registers. The language requires classes to be ordered sets of ordered sets. Classes simply partition the resources into sets of valid locations to place values. The outer set indicates the order to consider placing the arguments. In this example, when passing arguments in memory, we consider memory locations in low-to-high address order. The inner set typically contains a single element (the starting location). More complicated conventions make more use of the inner set.

The remaining lines (18–23) of the data transfer contain the argument placement description. The universal quantifier ($\forall$) operator iterates over the set, each time binding the variable argument to an element of the set. Here, the set is ordered, ensuring that argument will take values in the set in order. The resource ARG is a special resource that is provided by the outer environment. It contains information such as the type and size of the arguments for the call.

The two operators on line 19 complete the placement description. The placement operator ($\rightarrow$) is invoked for each value argument is assigned. The placement operator takes a value (here an argument) and a list of classes. The classes are searched, in order, for an available resource to place the given value. When a resource is found, the location is marked as used, by setting the assigned attribute, to ensure unique locations for each placed value. The selection operator ($\perp$) is used on labeled sets. This is simply a case expression. Based on the value of argument's type attribute, one expression from the labeled set is selected.

3.4 Point-of-view change

A view change indicates something has happened that caused values to *appear* to move. The register window mechanism on the SPARC microprocessor is an example.

When the register window slides, the contents of the registers appear to move because the names of the registers have changed. We wish to indicate this change without causing the move to occur. The change of view indicates how the names of locations have changed. View change is used more commonly when describing that a frame has been pushed on the stack. When a push occurs, all locations referenced by the stack pointer appear to shift.

4. THE FORMAL MODEL

4.1 P-FSA Representation

We use finite state automata to model each placement in a calling convention. One such FSA is shown in Fig. 5. This FSA models the placement of procedure arguments for the simple 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.

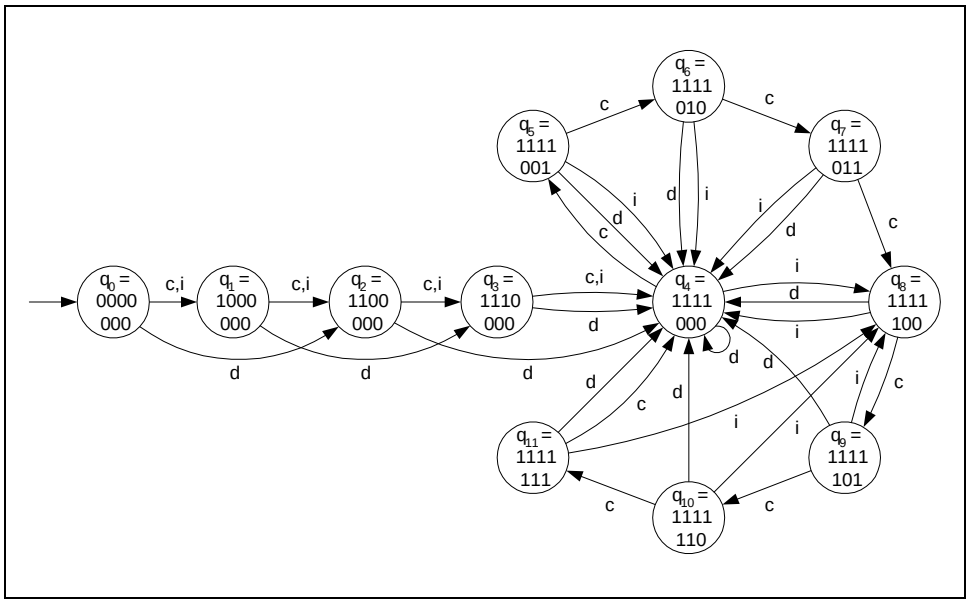

Fig. 5. P-FSA for transmission of parameters for a simple calling convention.

The states of the machine represent that state of allocation for the machine resources. For example, the state labeled q_2 represents the fact that registers a^1 and a^2 have been allocated, but that registers a^3, 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.

Modeling the allocation of an infinite resource, such as the stack, using an FSA poses a problem, however. As stated above, the state indicates 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 a complimentary problem. Since only the "alignment state" of the stack is of interest, structures that affect the state of the P-FSA differently must use different transitions. So for a convention that requires structures to be passed in 8-byte aligned memory locations, all structures of size n where n mod 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* $C = \{c_1, c_2, \ldots, c_m\}$ 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

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

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 [Mealy 1955]. We define a P-FSA, M, as a six-tuple[1] $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, 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 $\delta: Q \times \Sigma \rightarrow Q$,

— 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.

We also define $\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ and $\hat{\lambda}: Q \times \Sigma^* \rightarrow \Delta^*$ which are just string versions (defined by Hopcroft and Ullman [Hopcroft and Ullman 1979]) of δ and λ, respectively. So, for our example, we have $M = (Q, \{\text{char, int, double}\}, \{a^1, a^2, a^3, a^4\} \cup \{0, 1\}^3, \delta, \lambda, q_0)$, where Q and δ are pictured in Fig. 5 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 indicates 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	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	m_1[a]	m_2[b]	m_2	m_2	m_2	m_1	m_1	m_1
double	a^1a^2	a^2a^3	a^3a^4	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 phred(double, double, char, int);
```

will take the P-FSA in Fig. 5 from state q_0 to q_4 producing the string (a^1 a^2) (a^3 a^4) (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 the phred's arguments. The first double is placed in registers a^1 and a^2, the second in registers a^3

[1]In this paper, we use the notation of Hopcroft and Ullman [1979] for finite state automata and regular expressions. 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.

and a^4, the char at the stack location with offset zero and the int at the stack location with offset five.

4.2 Automatic P-FSA Construction

In this section, we present an algorithm for automatically constructing P-FSA's. For the moment, we assume the existence of a function $f:\Sigma^* \rightarrow \Delta^*$. f computes the same value as M. Since f and M are equivalent, why construct M at all? The answer is that f may have undesirable properties. For instance, M may be used in a context, such as a compiler, where performance is an issue. If f is implemented as an interpreter, the time it takes to compute a placement may not satisfy the performance constraints. Additionally, by using a P-FSA, there are several properties (such as an upper bound on M's execution time) we can prove about the P-FSA that we cannot prove about f. We present such properties in Section 4.3.

We construct the P-FSA by performing a depth-first-traversal of the states in Q to determine the set of reachable states from q_0. At each state q, the states that are reachable from q in one step are determined by using each element of $\{wc \mid c \in C\}$ as input to f. Each newly reachable state q' is added to Q and is subsequently visited by BUILD-P-FSA (the algorithms are included in the appendix). Finally, the appropriate additions to δ and λ are made for q'. BUILD-P-FSA also uses an auxiliary function STATE-LABEL:$P \rightarrow Q$. STATE-LABEL takes an output string from M and computes the label for the state that M was in when the input was exhausted.

Our construction is now complete, except the definition of the function f. We supply f's definition using an interpreter. The interpreter takes as input a CCL specification, information about a procedure's signature and some additional information about the target machine, and produces the necessary mapping information to properly call the given procedure. Thus, this interpreter can be used to implement f in our algorithm above. In Section 5.1, we present the interpreter's use in an implementation.

4.3 Completeness and Consistency in P-FSA's

Applications, such as compilers and debuggers, which generate, or process procedures at the machine-language level require knowledge of the calling convention. Until now, the portion of such an application's implementation that concerned itself with the procedure call interface was constructed in an *ad-hoc* manner. The resulting code is complicated with details, difficult to maintain, and often incorrect. In our experience, we have encountered many recurring difficulties in the calling convention portion of a retargetable compiler. 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 Fig. 2). 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 stem 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 that 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 tackled. During this process, some aspect of the first stage may break due to the interactions between the two pieces.

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

(1) Completeness:
   (a) *Does the specified convention handle any number of arguments?*
   (b) *Does the convention handle any combination of argument types?*

(2) Consistency:
   (a) *Does the convention map more than one argument to a single machine resource?*
   (b) *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 CCL specifications that ensure desirable responses to the above questions.

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 $c \in C$.

Using P-FSA's, we can guarantee that no incomplete convention will go undetected. For an incomplete convention K to not be detected, it would first have to be constructed using our algorithm. Assume such a P-FSA M exists for K. Then there must be some state q_k that is reachable from q_0 but does not have $\delta(q_k, a)$ defined for some $a \in C$. Let W_k denote the set of all strings x such that $\delta(q_0, x) = q_k$. That is, W_k is the set of strings that take M from state q_0 to q_k. Thus, for all strings x such that $x \in W_k$, xa

represents a signature that K does not cover. However, during construction, BUILD-P-FSA visited state q_k with some string w such that $\delta(q_0, w) = q_k$. Thus, w must be in W_k and must not be covered by K. Since BUILD-P-FSA calls $f(wc)$ for all $c \in C$, f will be called using $f(wa)$. Since wa is not covered by K, $f(wa)$ will be undefined. At this point the construction process will signal that K is incomplete.

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 ($\wedge$) 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.

5. USE IN A COMPILER

In this section, we present how the information from our CCL descriptions can be used to generate calling sequences for the *vpcc/vpo* optimizing compiler.

5.1 The Interpreter

We have implemented an interpreter for the CCL specification language. The interpreter's source is approximately 2500 lines of Icon code [Griswold and Griswold 1990]. The interpreter takes as input the CCL description of a procedure calling convention, a procedure's signature, and some additional information about the target architecture, and produces locations of the values to be transmitted, in terms of both the callee and the caller's frame of reference.

We have developed CCL specifications for the following machines: MIPS R3000, SPARC, DEC VAX-11, Motorola M68020, and Motorola M88100. Each of these CCL specifications is approximately one page in length. Using the specification for the MIPS, and the CCL interpreter, we constructed a P-FSA that implements the MIPS calling convention. The MIPS P-FSA uses only 70 out of a possible 512 states (the state label has nine bits), but requires up to 25 transitions for each state to implement the selection criteria for the **C** programming language. Since the MIPS convention has more machine resource classes and alignment requirements than any of the other machines, it represents the most complicated convention we have. For machines that pass procedure arguments on the stack with no alignment restrictions, such as the VAX-11, the FSA's contain only a few states.

For comparison purposes, we have examined the calling convention specific code for a retargetable compiler. The MIPS implementation requires 781 lines of **C** code, while the SPARC implementation has 618 lines. This code is one of the most complex sections of the machine-dependent code. This code is replaced by the P-FSA tables and a simple automaton interpreter.

5.2 Realizing the Calling Sequence

In our compiler, the code for the procedure bodies is generated without knowledge of the calling convention. For a callee, the optimizer treats formal parameters as local variables. It assigns each parameter either a register or a memory location, based on the parameter's predicted reference frequency. Thus, although an established convention for where values cross the procedure call interface exists, the code generated by our compiler for a procedure's body may not conform to the convention.

To correct this problem, instructions are placed before and after the callee's body, and before and after the call site in the caller. We call these instructions the caller/callee prologue/epilogue sequences. It is these sequences of instructions that are collectively called the calling sequence. The sequences introduce four new interfaces shown in Fig. 6. In each sequence, the instructions transform a convention interface to a code body interface or vice versa. Since these sequences of instructions are used to "glue" the procedure bodies to the convention interfaces, they correspond to the agents, shown in Fig. 3, of our high-level model.

An agent's responsibilities fall into one of three categories: allocation or deallocation of storage space, movement of values from their locations in the first interface to locations in the second interface, and the construction/restoration of procedure execution environments. Hence, to generate an agent's actions, we must have information about where the calling convention expects values, what space to allocate or free, and the procedure's environment structure. We can automatically generate the first two.

To illustrate our technique, we show how to generate the instruction sequence for one agent. The instruction sequences that correspond to the other three agents are generated exactly the same way. For our example, we focus on the prologue callee agent for the procedure foo introduced earlier.

Recall that for our hypothetical machine, foo's arguments are placed by the caller in locations a^1, a^2, a^3, M[sp:sp+7]. Assume that in generating foo's body, the optimizer uses two persistent registers, allocates 12 bytes of memory for local variables (includ-

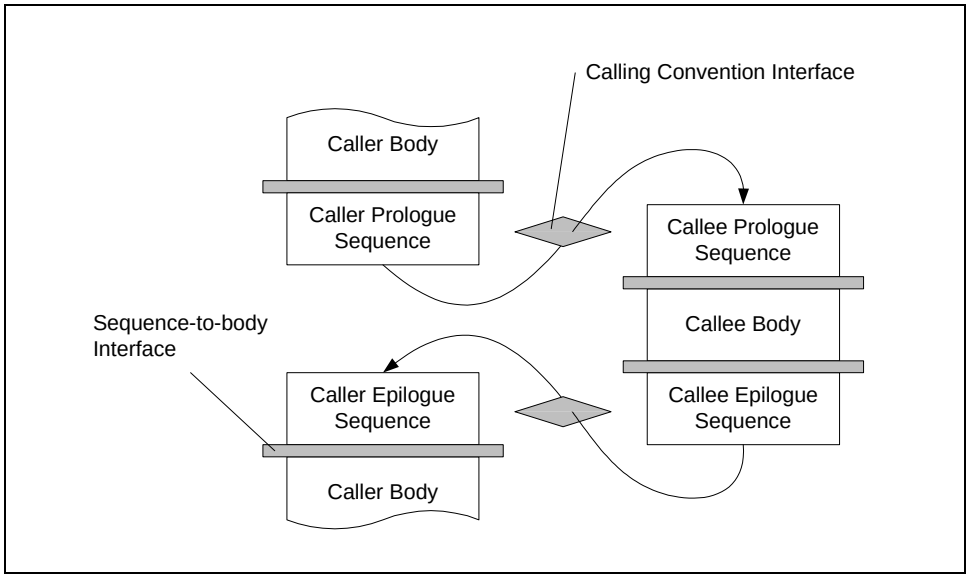

Fig. 6. Calling sequence locations.

ing foo's arguments) and uses eight bytes of spill space. One possible frame layout is shown in Fig. 7. Fig. 7a shows the generic layout for any procedure, while Fig. 7b shows foo's layout using this scheme. The relative locations of the temporary spill space, local variable space and persistent register save space are determined by the optimizer. The optimizer provides the locations where the callee body expects values. These are listed in the second column of Table II. These locations represent an agreement between the callee body and the callee prologue agent.

The optimizer calls the P-FSA interpreter with foo's signature and values of the external variables:

[SPILL_SIZE=8, LOCALS_SIZE=12, NVSIZE=8,
(**ARG**^{1}, type:char, size:1), (**ARG**^{2}, type:int, size:4), (**ARG**^{3}, type:int, size:4),
(**ARG**^{4}, type:double, size:8)]

The P-FSA returns view changes, a list of argument locations that correspond to the calling convention, and a list of persistent registers:

[($\forall$ *offset* $\in$ {-∞:∞}, **M**[sp + *offset*] : **M**[sp + *offset* + 32]),
[(**ARG**^{1}, **a**^{1}), (**ARG**^{2}, **a**^{2}), (**ARG**^{3}, **a**^{3}), (**ARG**^{4}, **M**[sp+32:sp+39])],
[persistent**:** **a**^{6}, **a**^{7}, **a**^{8}, **a**^{9}]]

In this example, the view change occurred before the list of locations. Therefore, the locations reflect this fact.

View change information corresponds to the allocation or deallocation of storage space. This view change indicates that any memory location's address that contains a valid value for offset, shifts down by 32 bytes. Since offset can take on any positive or negative value (-∞:∞), this corresponds to all addresses relative to the stack pointer. Thus, a decrement of the stack pointer by 32 bytes is needed. This allocation of stack

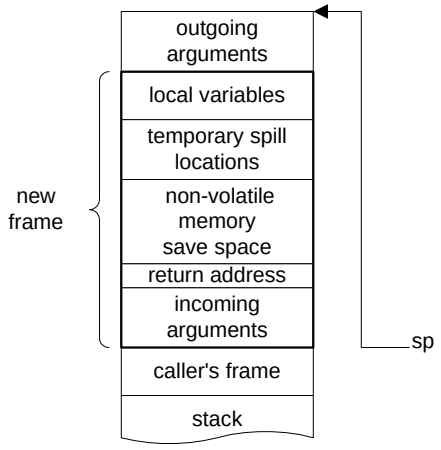

Fig. 7a. Generic Frame Layout.

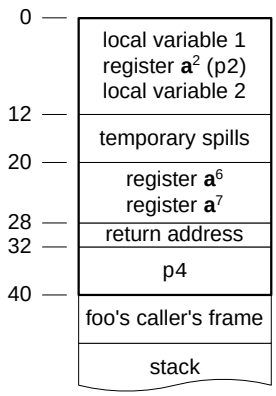

Fig. 7b. foo's Frame Layout.

Fig. 7. A possible procedure activation frame structure.

space will appear as a view change since it changes the names of all locations referenced by the stack pointer. A table is consulted for each view change in the CCL description. The table maps all view changes to valid machine instructions.

After the view change has been performed, the necessary moves must be made to transform the agreement between the caller prologue agent and callee prologue agent to the agreement between the callee prologue agent and the callee body. Table II summarizes the location information. Column one shows the locations returned by the P-FSA. Column two shows the locations that the optimizer supplies. Column three, which can be trivially derived from columns one and two, indicates the necessary actions. Each of these moves is a register/memory to register/memory move. A table of available move instructions is consulted to determine the necessary instructions to be inserted into the callee prologue's sequence.

After the agent's actions are determined, the list of sources and destinations must be examined to determine if there are any dependencies. If a source is also a destination, the move containing the source must be performed before the move containing the destination, otherwise the source value will be lost. It is not uncommon for a circularity to exist. For example, if $a^1 \rightarrow a^2$ and $a^2 \rightarrow a^1$, we must introduce a third location to break the circularity: $a^1 \rightarrow temp$, $a^2 \rightarrow a^1$, $temp \rightarrow a^2$. Either an available register or a memory location must be used to temporarily hold one of the values. In our compiler, we usually have a register available.

At this point, the callee prologue instruction sequence is complete. So far, we have not addressed instruction sequence efficiency. Because of the frequency of procedure calls, generating efficient instruction sequences is an important feature of optimizing compilers. In our compiler, the resulting instruction sequences are processed by the optimizer. Thus, although the instruction sequences that are initially generated by this

	Convention	Callee Prologue Agent/ Callee Agreement	Callee Prologue Agent Actions
Arguments	p1:$\mathbf{a}^1$	p1:$\mathbf{a}^3$	$\mathbf{a}^1 \rightarrow \mathbf{a}^3$
	p2:$\mathbf{a}^2$	p2:**M**[sp+4:sp+7]	$\mathbf{a}^2 \rightarrow$ **M**[sp+4:sp+7]
	p3:$\mathbf{a}^3$	p3:$\mathbf{a}^4$	$\mathbf{a}^3 \rightarrow \mathbf{a}^4$
	p4:**M**[sp+32:sp+39]	p4:$\mathbf{a}^1$,$\mathbf{a}^2$	**M**[sp+32:sp+39]$\rightarrow \mathbf{a}^1,\mathbf{a}^2$
Persistent	$\mathbf{a}^6$	**M**[sp+20:sp+23]	$\mathbf{a}^6 \rightarrow$ **M**[sp+20:sp+23]
	$\mathbf{a}^7$	**M**[sp+24:sp+27]	$\mathbf{a}^7 \rightarrow$ **M**[sp+24:sp+27]
	$\mathbf{a}^8$	$\mathbf{a}^8$	—
	$\mathbf{a}^9$	$\mathbf{a}^9$	—

Table II. Summary indicating how callee prologue agent actions are determined from placement information from both interfaces.

process are naive, they benefit from thorough optimization just as other code does. The resulting code is as good, if not better, than the code generated by our handwritten version of our compiler. Often, the code improves because the additional peephole optimization phase that is performed after the calling sequence instructions are generated can remove unnecessary register-to-register moves.

6. CONSTRUCTION OF DIAGNOSTIC PROGRAMS

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. In this section, we discuss how CCL descriptions can be used to make compilers more robust without requiring that the compiler's implementation use CCL.

6.1 Test Vector Selection

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 [Hennie 1964; Kohavi 1978]. 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 III 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 important, 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.

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 III. P-FSA profiles for several calling conventions.

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 Fig. 5, 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);
```

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

untested. Clearly such a test should be included in the suite. To further illustrate the problem, consider the FSA specification shown in Fig. 8a. An erroneous implementation, shown in Fig. 8b, 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.

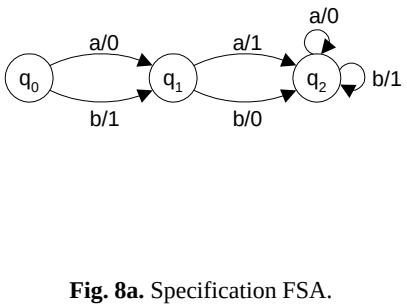

Fig. 8a. Specification FSA.

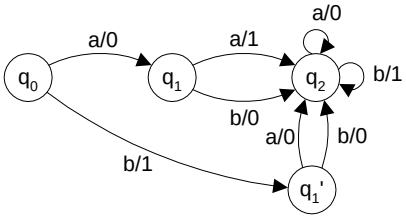

Fig. 8b. Implementation FSA.

Fig. 8. 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 Fig. 9, 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 Fig. 8. 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 Fig. 5, 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.

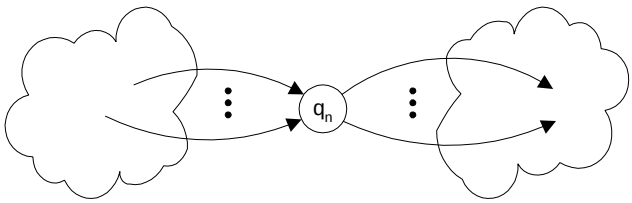

Fig. 9. Entering and exiting transitions for a state.

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.

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	$8x10^8$

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

An algorithm for generating transition-pair paths is shown in Fig. 15 (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.

6.2 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 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.

Fig. 10 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 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. Although this process uses two compilers, the same pro-

cess 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.

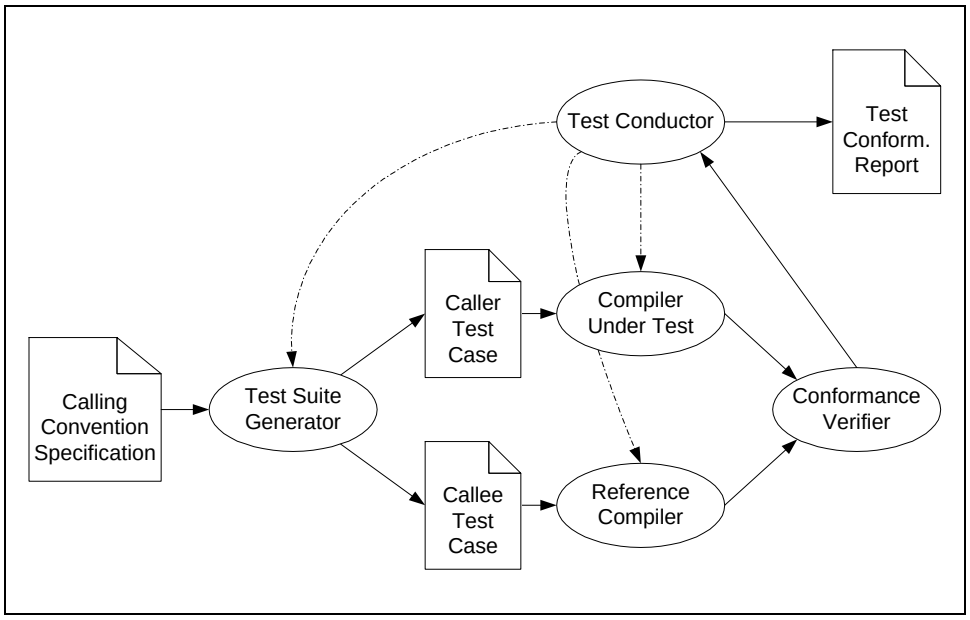

Fig. 10. The compiler conformance test process.

In each test case, 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 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 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.

6.3 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 above, 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. The result of running all four programs is called an *outcome*. Fig. 11 shows an outcome pictorially. Procedures generated by the reference compiler are filled, while CUT generated components are unfilled. The result of a single test is indicated by an arrow connecting a pair of components. When the result is that a test passed, a solid line is shown, while a dotted line is used for test failure.

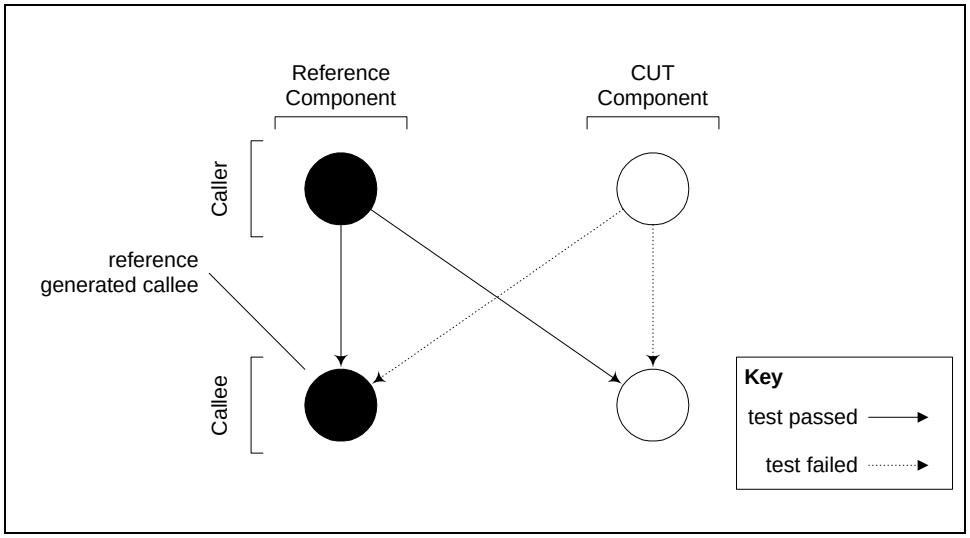

Fig. 11. An example outcome.

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 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 Fig. 11, the CUT/reference test (the test composed 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 fails, 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 composed 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 V 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 10 and 12 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.

6.4 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* [Fraser and Hanson 1991; Fraser and Hanson 1995] compiler, several versions of GNU's *gcc* [Stallman 1992], 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.

---

[1]Appel [1996] observes that such outcomes actually are possible. 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
	Faults in at least three components.		Faults in both components of the CUT.
	Faults in both components of reference compiler.		CUT implements wrong convention (not externally conformant with the reference).
	Fault in the reference compiler's caller. Fault in the CUT's callee.		Fault in the CUT's callee.
	Fault in the reference compiler's caller.		Not possible.
	Fault in reference compiler's callee. Fault in CUT's caller.		Fault in the CUT's caller.
	Fault in reference compiler's callee.		Not possible.
	Two conventions. One shared between the reference compiler's callee and CUT's caller, and vice versa.		Not possible.
	Not possible.		No faults detected.

Table V. 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 Fig. 12. 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.

The results of running both internal and external tests on the compiler set for the MIPS are shown in Table VI. We found both internal and external conformance errors in all of the tested compilers. Table VI reports internal and external errors separately. Within each class, the number of actual tests that failed and the number of faults that

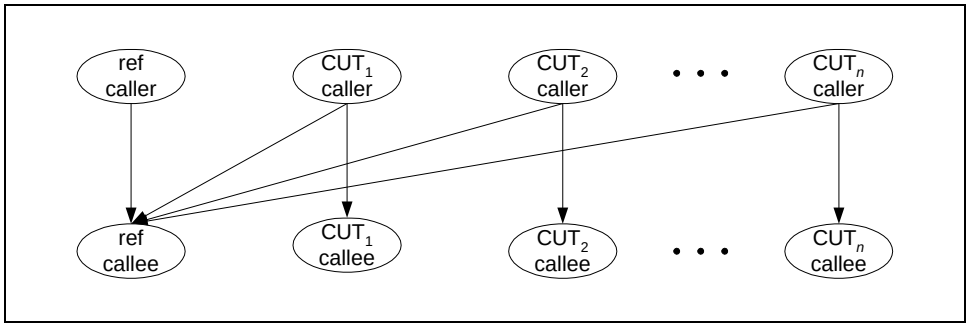

Fig. 12. Determining conformance of *n* compilers.

	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 VI. 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.

caused failure are indicated[1]. 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.

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:

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

```
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 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);
```

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 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

---

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

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.

7. RELATED WORK

What little work there has been in calling sequences has been *ad-hoc*. For example, Johnson and Richie discuss some rules of thumb for designing and implementing a calling sequence for the **C** programming language [Johnson and Ritchie]. Davidson and Whalley [1991] experimentally evaluated several different **C** calling conventions. However, no attempts have been made to formally analyze calling conventions.

On the other hand, the use of FSA's for modeling parts of a compiler, and as an implementation tool, has a long and successful history. For example, Johnson et al. [1968] describe the use of FSA's to implement lexical analyzers. More recently, Proebsting and Fraser [1994], and Muller [1993] have used finite state automata to model and detect structural hazards in pipelines for instruction scheduling.

Work related to the automatic generation of test suites has received much attention recently in the area of conformance testing of network protocols [Sidhu and Leung 1989]. The purpose of the suite 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 [Fujiwara et al. 1991], Distinguishing Sequence Method [Kohavi 1978], and Unique-Input-Output method [Aho et al. 1991]. These methods are derivatives of the checking experiment problem where an implementation is checked against a specification FSM [Yannakakis and Lee 1995].

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 state machine 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. Finally, a similarly related field is the automatic verification of digital circuits [Hennie 1964; Ho et al. 1995].

8. CONCLUSION

Current methods of procedure calling convention specification are frequently imprecise, incomplete, or contradictory. This comes from the lack of a formal model, or specification language that can guarantee completeness and consistency properties. We have presented a formal model, called P-FSA's, for procedure calling conventions that can ensure these properties. Furthermore, we have developed a language and interpreter for the specification of procedure calling conventions. With the interpreter, 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 we have shown that it is possible to automatically generate the calling sequence generator of a compiler, some work is required to retrofit an existing compilation system to use CCL descriptions. Fortunately, it is possible to reap the benefits of CCL 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 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 faults in calling sequence generators of mature compilers. Undoubtedly, it would be even more effective during the initial development of a compilation system.

9. REFERENCES

Aho, A. V., Dahbura, A. T., Lee, D., and Uyar, M. U. (1991). 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.

Appel, A. W. (1996). Personal Communication.

Bailey, M. W. and Davidson, J. W. (1995). A formal model and specification language for procedure calling conventions. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298–310.

Bailey, M. W. and Davidson, J. W. (1996a). Reusable Application-Dependent Machine Descriptions. In *Workshop Record of The Inaugural Workshop on Compiler Support for Systems Software*, pages 77–85.

Bailey, M. W. and Davidson, J. W. (1996b). Target-Sensitive Construction of Diagnostic Programs for Procedure Calling Sequence Generators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257.

Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338.

Benitez, M. E. and Davidson, J. W. (1994). The advantages of machine-dependent global optimization. In *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, pages 105–124.

Davidson, J. W. and Whalley, D. B. (1991). Methods for saving and restoring register values across function calls. *Software–Practice and Experience*, 21(2):149–165.

Digital Equipment Corporation (1978). *VAX Architecture Handbook*. Digital Equipment Corporation.

Digital Equipment Corporation (1993). *Calling Standard for AXP Systems*. Digital Equipment Corporation, Maynard, MA.

Fraser, C. W. (1993). Personal Communication.

Fraser, C. W. and Hanson, D. R. (1991). A code generation interface for ANSI C. *Software–Practice and Experience*, 21(9).

Fraser, C. W. and Hanson, D. R. (1995). *A Retargetable C Compiler: Design and Implementation*, Benjamin Cummings.

Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603.

Griswold, R. E. and Griswold, M. T. (1990). *The Icon Programming Language*. Prentice-hall, second edition.

Hennie, F. C. (1964). Fault detecting experiments for sequential circuits. In *Proceedings of the Fifth Annual Symposium on Switching Theory and Logical Design*, pages 95–110.

Ho, R. C., Yang, C. H., Horowitz, M. A., and Dill, D. L. (1995). Architecture validation for processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 404–413.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Johnson, S. C. A tour through the portable C compiler.

Johnson, S. C. and Ritchie, D. M. The C language calling sequence. Bell Labs.

Johnson, W. L., Porter, J. H., Ackley, S. I., and Ross, D. T. (1968). Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813.

Kane, G. and Heinrich, J. (1992). *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language*. Prentice-Hall.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice-Hall, second edition.

Kohavi, Z. (1978). *Switching and Finite Automata Theory*. McGraw-Hill, second edition.

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

Muller, T. (1993). Employing finite automata for resource scheduling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 12–20.

Proebsting, T. A. and Fraser, C. W. (1994). Detecting pipeline structural hazards quickly. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286.

Sidhu, D. P. and Leung, T.-K. (1989). Formal methods for protocol testing: A detailed study. *IEEE Transactions of Software Engineering*, 15(4):413–426.

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

Yannakakis, M. and Lee, D. (1995). Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50:209–227.

Appendix

This appendix presents the supporting algorithms discussed earlier. It also contains a more sophisticated example specification for the MIPS R3000.

Construction Algorithms

We define the algorithm BUILD-P-FSA in Fig. 13. The algorithm starts with the initial state q_0 as the only element of Q. Since there are no transitions yet, λ and δ have no rules. A call to BUILD-P-FSA takes three parameters, q, w, and x. q represents the state for BUILD-P-FSA to visit, while w represents the input string such that (q_0, w) yields (q, ε), and x is output string upon reaching q. From this definition, the initial call to BUILD-P-FSA must be BUILD-P-FSA(q_0, ε, ε).

```
function BUILD-P-FSA(q, w, x)
    // q ∈ Q, w ∈ Σ*, x ∈ Δ* | λ̂(w) = x
    for each criterion c ∈ C do
        y ← f(wc);                          // compute placement of signature wc
        q' ← STATE-LABEL(y);                // compute state label from placement
        if q' ∉ Q then
            Q ← Q ∪ {q'};
            BUILD-P-FSA(q', wc, y);
        end if
        a ← b | xb = y;                     // set a as the suffix of y not in x
        add λ(q, c) = q';
        add δ(q, c) = a;
    end for
end function
```

Fig. 13. Algorithm to build a P-FSA.

The algorithm for STATE-LABEL is simple. We start with state q_0. As STATE-LABEL reads each symbol from the string, it encounters either the name of a finite resource, or a symbol representing the distinguishing bits of p. In the finite case, the bit corresponding to the resource is set in the finite resource vector. In the infinite case, the distinguishing bits of the state are set to the input symbol that was read. At the end of the input, all finite resources that have been read have their bits set to indicate they are unavailable, and the distinguishing bits indicate the last set of distinguishing bits read. To complete the computation, we need to move the infinite resource index to the next available resource (it currently points to the last unavailable one)[1]. The result of this computation is precisely the label for the final state of M for output w since it indicates which resources are available for allocation. The complete algorithm is shown in Fig. 14.

[1]An ordered list of values for p's distinguishing bits is known so that we can perform this calculation, although this is usually just an increment.

```
function STATE-LABEL(w)                          // w ∈ Δ*
    z ← 0^n;                                     // z is the finite resource vector
    while w ≠ ε do
        // extract the first symbol from w
        define a and x such that ax = w;
        w ← x;                                   // set w to the rest of w
        if a ∈ R then                            // for finite resources:
            // mark it as used
            set a's corresponding bit in z;
        else                                     // for infinite resources:
            d ← a;                               // keep the last one encountered
        end if
    end while
    d ← d + 1;                                   // set d to the next resource
    return zd;                                   // return state label made of z and d
end function
```

Fig. 14. Definition of STATE-LABEL.

```
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 δ(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
                                                     // Compute paths from here
            P ← TRANSITION-PAIRS(δ(q, a), wa, V ∪ {q}, cycle);
            paths ← paths ∪ P;
        end if
        paths ← paths ∪ {wa};                        // Add this path to paths
    end for
    return paths;                                    // Return paths from q
end function
```

Fig. 15. Test vector generation algorithm.

A.2 A Complex Example

We now present a significantly more complex example: the MIPS R3000. The MIPS is a RISC machine with both integer and floating-point registers. Unlike most machines, the MIPS convention designates that not only some integer registers but also some floating-point registers are to be used for passing arguments. Fig. 16 contains the complete convention specification.

```
1   external SPILL_SIZE, LOCALS_SIZE
2   persistent {r^1, r^16:23, r^26:31}
3   alias REG_ARGS ≡ 16
4   alias sp ≡ r^29
5   caller prologue
6       view change
7           ∀ offset ∈ {−∞:∞}
8               M[sp + offset] becomes M[sp + offset + ⌈ARG_BLOCK_SIZE⌉^8]
9       end view change
10      data transfer (asymmetric)
11          alias rindex ≡ 4:7
12          alias fpindex ≡ 12,14
13          alias mstart ≡ sp + REG_ARGS
14          alias mindex ≡ mstart:∞
15          resources {<r^rindex, M^mindex>, <f^fpindex, M^mindex>, <M^mindex>}
16          ∀ mem ∈ {M[sp(REG_ARGS)]} set mem.assigned ← true
17          internal ARG_BLOCK_SIZE ← ∑(<M[addr].size | addr ∈ <mindex>
18              ∧ M[addr].assigned>)
19          class intregs ← <<r^x>| x ∈ <rindex>>
20          class intfpregs ← <<r^x> | x ∈ <rindex> ∧ x mod 2 = 0>
21          class fpfpregs ← <<f^x> | x ∈ <fpindex>>
22          class mem ← <<M^loc> | loc ∈ <mindex> ∧ loc mod 4 = 0>
23          class aligned_mem ← <<M^loc> | loc ∈ <mindex> ∧ loc mod 8 = 0 >
24          class struct_mem ← <<r^x, M^mstart> | x ∈ <rindex>>
25          class aligned_struct_mem ← <<r^x, M^mstart> | x ∈ <rindex> ∧ x mod 2 = 0>
26          ∃ reg ∈ {reg | reg ∈ {f^12} ∧ reg.assigned} ⇒ set r^4.unavailable ← true
27          ∃ reg ∈ {reg | reg ∈ {f^12} ∧ reg.assigned} ⇒ set r^5.unavailable ← true
28          ∃ reg ∈ {reg | reg ∈ {f^14} ∧ reg.assigned} ⇒ set r^6.unavailable ← true
29          ∃ reg ∈ {reg | reg ∈ {f^14} ∧ reg.assigned} ⇒ set r^7.unavailable ← true
30          ∃ reg ∈ {reg | reg ∈ {r^4:5} ∧ reg.assigned} ⇒ set f^12.unavailable ← true
31          ∃ reg ∈ {reg | reg ∈ {r^6:7} ∧ reg.assigned} ⇒ set f^14.unavailable ← true
32          ∀ argument ∈ <ARG^1:ARG_TOTAL>
33              map argument → argument.type ⊥ {
34                  byte, word, longword: <intregs, mem>,
35                  struct:         argument.size ⊥ {
36                      1,2,3,4,5,6,7:      <struct_mem, mem>,
37                      default:            <aligned_struct_mem, aligned_mem>
38                  },
39                  float, double:      ARG^1.type ⊥ {
40                      struct, byte, word, longword:   <intfpregs, aligned_mem>,
41                      float, double:                  <fpfpregs, aligned_mem>
42                  }
43              }
44      end data transfer
45  end caller prologue
```

Fig. 16. The MIPS R3000 specification.

```
46  callee prologue
47      view change
48          ∀ offset ∈ {-∞:∞}
49              M[sp + offset] becomes M[sp + offset + ⌈SPILL_SIZE +
50                  LOCALS_SIZE + NVSIZE⌉8]
51      end view change
52  end callee prologue
53  callee epilogue
54      data transfer (asymmetric)
55          resources {<r2>,<f0>}
56          ∃ return ∈ <RVAL1:RVAL_TOTAL> ⇒
57              map return → return.type ⊥ {
58                  byte, word, longword:   <<<r2>>>,
59                  float, double:          <<<f0>>>,
60                  struct:                 ↑(<<<r2>>>)
61              }
62      end data transfer
63  end callee epilogue
```

Fig. 16. The MIPS R3000 specification.

Although the MIPS convention is more complicated, the description is quite similar to our previous example—with a few additional restrictions. First, notice that the resource list (line 15) now includes the floating-point registers. Each resource set is ordered to indicate that the resources within them must be assigned in sequence. This prevents the subsequent placement operator from using element *n* after element *n* + 1 has been assigned. Second, we have added several new classes. These reflect the addition of registers for passing arguments and alignment constraints placed on the registers and stack. For example, the class ‘intfpregs’ is the set of starting points in the integer register set that have even register numbers used for passing floating-point values. The class ‘aligned_mem’ is the set of stack locations that are 8-byte aligned. Finally, the class ‘struct_mem’ contains a set of starting-point pairs. The pair is used to indicate that if the placement depletes the first resource, the placement continues using the second resource starting point. This class is used in passing structure arguments and indicates that a single structure argument may span the argument registers and stack.

After properly defining the classes, the placement (lines 32–43) is straightforward. For each type, a list of classes to use is specified. In each case, a register class is first, followed by the corresponding stack class. This reflects the convention that registers are used until depleted, followed by stack use. The placement is slightly complicated in the floating-point case since the register class to use is dependent on the type of the first argument. When the first argument is a floating-point value, the floating-point registers are used. When the first value is any other type, the integer registers are used to pass floating-point values.

The MIPS convention has two other features we must convey. The first requires that the initial 16 bytes of the frame, which correspond to the argument registers, must be reserved so the callee can save the register arguments if necessary. This is specified on line 16 by setting the ‘assigned’ attribute for these resources. The second constraint is that floating-point argument registers are associated with the integer registers (f^6 with r^4 and r^5, f^7 with r^6 and r^7). The association requires that if a register in one class is assigned, the associated register in the other class cannot be assigned. Each of the four associations is specified, on lines 26–31, using the existential quantifier (∃) which is simply a conditional expression. These restrictions complete the calling convention for the MIPS. The remaining details are similar to the simple example presented earlier.