Producing an Accurate Call-Stack Trace in the Occasional Absence of Frame Pointers

Max Copperman

UCSC-CRL-92-25
Supersedes UCSC-CRL-90-62
March 1992

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

An interactive debugger should be able to provide a call-stack trace, listing active subroutines in reverse order of their invocations. This facility relies on information provided by code within each called routine. This paper describes alternative ways to support this facility in the circumstance that this code is optimized away.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging — *debugging aids*; D.2.6 [Software Engineering]: Programming Environments; D.3.4 [Programming Languages]: Processors — *code generation, compilers, optimization*
General Terms: Algorithms, Languages
Additional Keywords and Phrases: debugging, compiler optimization, call-stack trace, run-time stack

1 Introduction

An interactive debugger has the capability of setting a breakpoint in a program. When a breakpoint is reached and the debugger takes control, presumably the user wishes to examine the state of the program. Part of the state that the user may wish to examine is the current execution context. Debuggers have the facility to provide a call-stack trace, which is a list of active routines and their arguments, in reverse order of invocation.[1] This facility relies on information provided by code within each active routine. If this code is eliminated due to optimization, the call-stack trace will contain incorrect and incomplete information, which may mislead the user. It would be preferable to provide either an admittedly incomplete stack trace containing only correct information or a stack trace that is identical to that which would be produced if the optimization had not been performed.

Providing a call-stack trace requires the ability to locate the callers's stack frame and symbol table information given the called routine's stack frame. Typically a call-stack trace displays the arguments of active routines but does not display local variables. A debugger may also allow the user to change the apparent context of execution so that any chosen active routine can be treated as the focus of debugging. The user can request the display or modification of arguments or local variables of the routine that is the focus of debugging. To fulfill such requests, the debugger needs the same capability that it needs to provide a call-stack trace: the ability to locate the callers's stack frame and symbol table information given the called routine's stack frame. Subsequently, only providing a correct call-stack trace is discussed. In particular, we do not further discuss the display of local variables.

This problem has not been not discussed in the literature. [Zel84] provides a solution to the problem of providing a correct call-stack trace in the presence of procedure integration (inlining).

This paper shows how an optimizing compiler and interactive debugger can cooperate so that the debugger can provide an accurate call-stack trace while making subroutine calls as inexpensive as possible.

2 The Call-Stack Trace

In order to provide a call-stack trace, a debugger uses information that is provided by the standard code sequence that implements a subroutine call, termed *the calling sequence*. In the presence of an optimization of the calling sequence, some of the information currently used by debuggers will be missing, causing the debugger to provide an incorrect call-stack trace.

Terminology

At any point in an executing program, some sequence of routines is active. The naming convention used within this paper for such a sequence is F_0, F_1, ..., F_n where F_0 is the first routine called and F_n is the currently executing routine. Thus for an arbitrary active routine F_i, its caller is F_{i-1} and the active routine that it called is F_{i+1}.

In the figures, *ip* denotes the instruction pointer register, *fp* denotes the frame pointer register, and *sp* denotes the stack pointer register,

2.1 The Calling Sequence

In a running program, the currently executing routine must have access to its arguments, the return address, local variables, and compiler temporaries. The standard method in many of today's machine architectures for providing such access is to provide storage for each active routine on a procedure call stack. The storage associated with the routine is known as the routine's *stack frame*. A machine register contains

[1] Call-stack trace is one of many terms used to describe this list of active routines. Others include stack trace, stack dump, procedure traceback and backtrace. We have chosen the term call-stack trace in the face of a lack of unanimity of usage.

a pointer to the base of the currently executing routine's stack frame, and the arguments and local variables are accessed as offsets from that pointer. The pointer is called the *frame pointer*, and the machine register that by convention contains the frame pointer is called the *frame pointer register* (this name is sometimes shortened to *frame pointer* as well – context is used to distinguish the two).

When one routine F_{i-1} calls another routine F_i, F_{i-1} (typically) pushes F_i's arguments on the stack. The call instruction itself pushes the return address on the stack. Code within F_i, called F_i's *prologue*, gets the machine ready for the body of F_i to execute. This includes (but is not limited to) making space for F_i's local variables and providing access to F_i's local variables and parameters. Access to F_i's local variables and parameters is provided by setting the frame pointer register to point to F_i's stack frame. However, after F_i has completed and control has been returned to F_{i-1}, the frame pointer register must contain F_{i-1}'s frame pointer. F_{i-1}'s frame pointer is therefore saved in F_i's stack frame prior to modifying the frame pointer register to point to F_i's stack frame. Space for F_i's locals is allocated on the stack by adjusting the stack pointer. Immediately prior to returning, code within F_i pops F_{i-1}'s frame pointer from the stack into the frame pointer register. See Figure 2.1 for an example of a standard calling sequence.

2.2 Optimization of the Calling Sequence

Under some circumstances, the code that pushes F_{i-1}'s frame pointer and sets the frame pointer register to point to F_i's stack frame is unnecessary overhead and can be eliminated. Figure 2.2 gives an example of a calling sequence upon which this optimization has been performed. If F_i is optimized in this manner and F_{i+1} is not, F_{i+1} will save F_{i-1}'s frame pointer, not F_i's as in the unoptimized case. See Figure 2.3 for an example of such a situation.

This optimization is possible when the frame pointer register is not used in F_i's code. This register is used for two purposes:

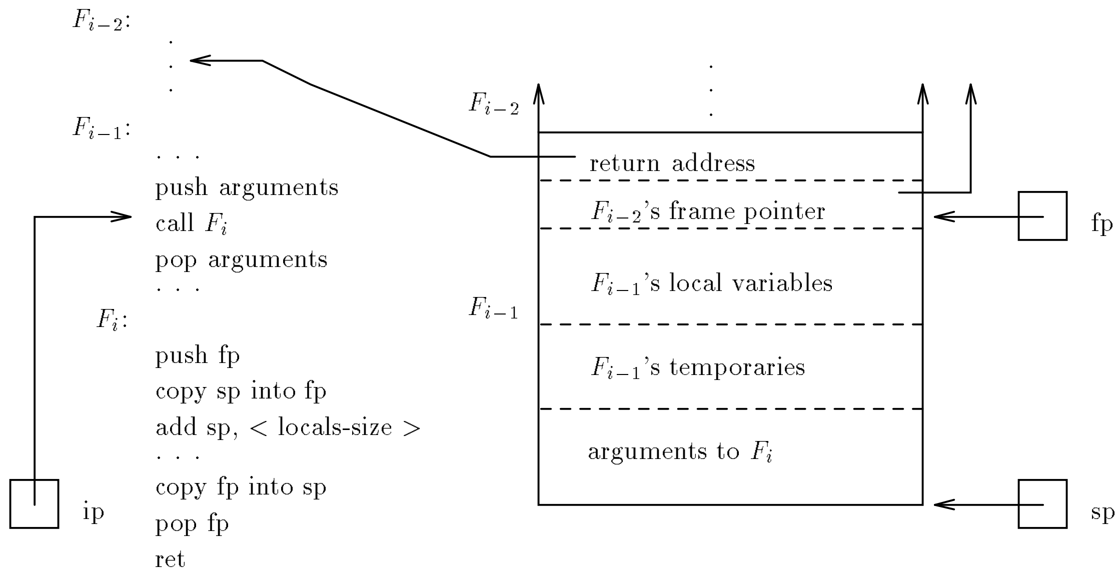
1. to access F_i's arguments and local variables, and
2. to restore the stack pointer to the position following F_{i-1}'s return address (in preparation for the execution of F_i's return instruction).

Clearly, if F_i has no arguments or local variables, or has them but does not reference them,[2] it will not use its frame pointer for the first purpose. Even if F_i does reference its locals or arguments, a compiler often has enough information to reference them through the stack pointer rather than the frame pointer, although doing so may add to the complexity of the compiler. In addition, a compiler often has enough information to restore the stack pointer to the position following F_{i-1}'s return address without using the frame pointer. Correct code for F_i can be produced without saving F_{i-1}'s frame pointer unless at some instruction within F_i, F_i's frame is not constant in size across all calls to F_i. This is infrequent, but happens if the stack is used for dynamic allocation or if the stack pointer is modified along one execution path but not along another.[3] Note that the compiler cannot reference local variables and arguments through the stack pointer in the same circumstances that it cannot to restore the stack pointer to the position following F_{i-1}'s return address without using the frame pointer.

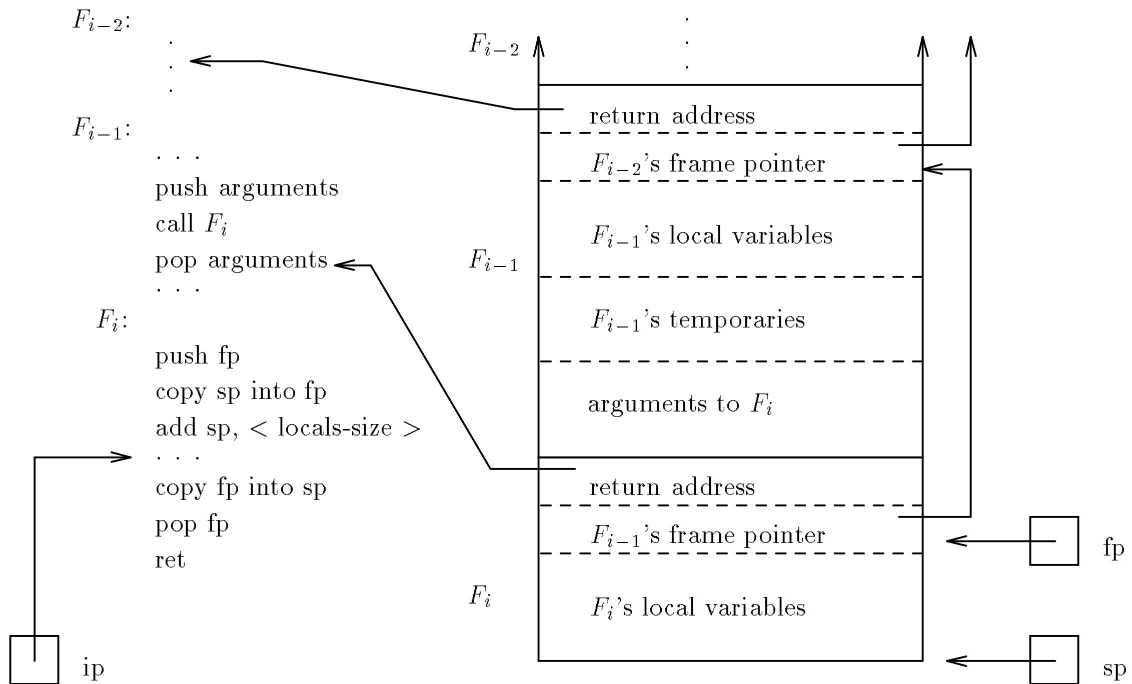
Such optimization of a routine's prologue and epilogue is most commonly done for routines that have neither parameters nor local variables, because less analyis is needed on the part of the compiler.

[2] A routine may have arguments and/or local variables but not access them due to carelessness, consistency requirements on a set of routines, or the use of stub routines during the development process.

[3] One way that the stack pointer can be modified along one execution path but not along another is if parameters are not popped immediately following a call (a routine may be called on one path and not on another; its parameters affect the size of the stack). Another way is if storage for a (conditionally executed) local block is allocated when the block is entered but not deallocated when the block is exited. The stack pointer may not need to be adjusted for each such allocation or set of parameters because an unoptimized prologue restores the stack pointer by setting it to the value in the frame-pointer register.

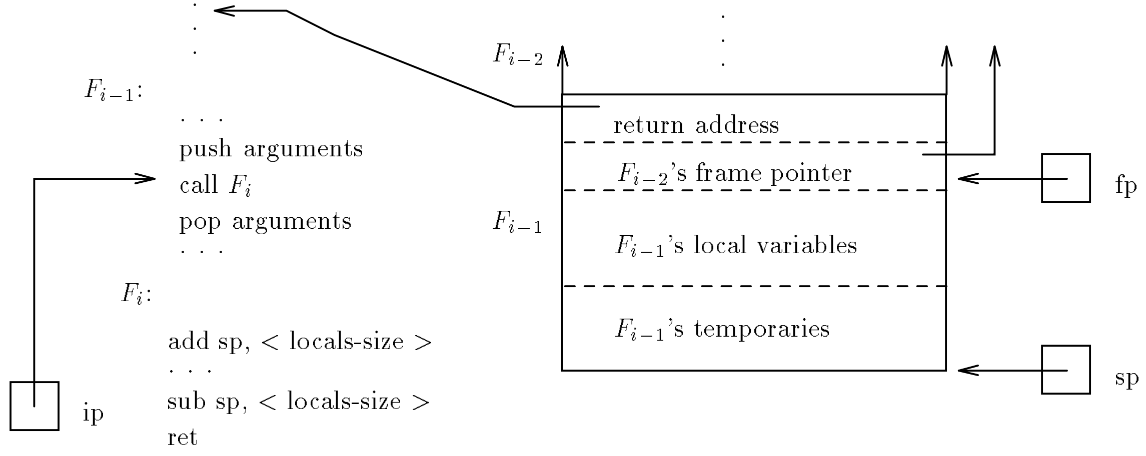


Just Before The Call

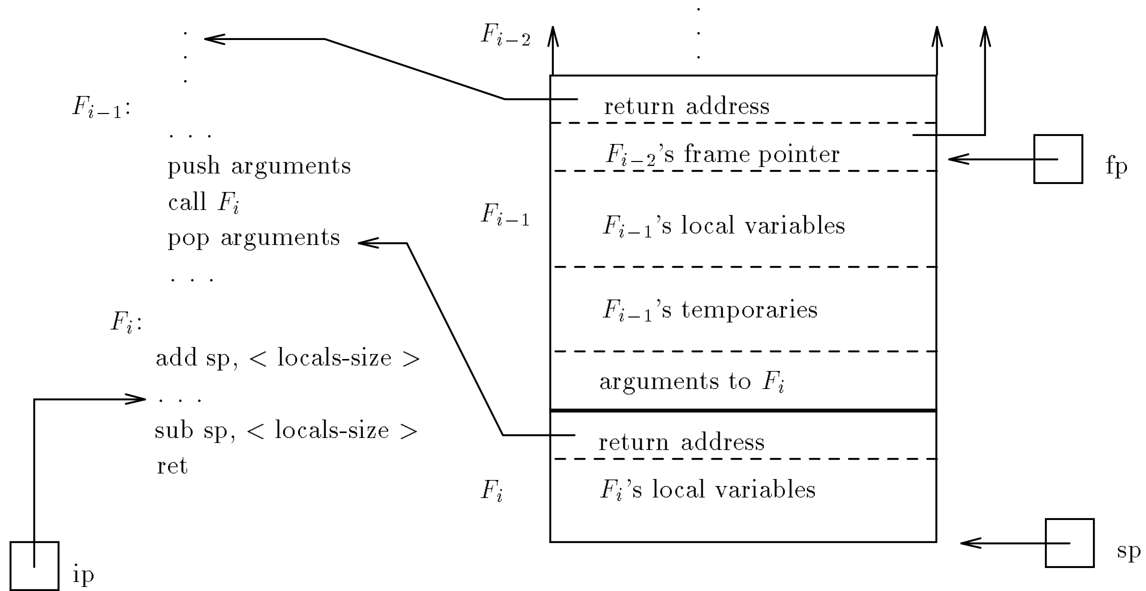


Calling Sequence Completed

Figure 2.1: Unoptimized Calling Sequence



Just Before The Call



Calling Sequence Completed

Figure 2.2: Optimized Calling Sequence

2.3 Debugger Use of Frame Pointers

The frame pointers that have been pushed onto the call stack by calling sequences form a linked list of pointers to active routine stack frames, with the value in the frame pointer register heading the list. In the unoptimized case, this list contains a pointer to the frame of every active routine, and the frame-pointer register points to the frame of the currently executing routine. A pointer to a routine's frame can be used to locate and access its arguments.

Given an address within a routine, the debugger can find the name and parameter list of the routine by looking in the symbol table. The code address used to find the symbol table entry for the currently executing routine is the address in the instruction pointer when the breakpoint is reached.[4] The code address used to find the symbol table entry for each other routine F_i is the return address stored in the stack frame of routine F_{i+1}.

Let us consider in detail how the debugger will construct the call-stack trace. In following the general description given here, it may be helpful to refer to the example call stack in Figure 2.1. The debugger begins with the currently executing routine F_n. An address within F_n is available from the instruction pointer. From that address, the parameter list and name of F_n are retrieved from the symbol table. The values of the arguments to F_n are available through the frame pointer register. They are displayed formatted according to the type information in the parameter list retrieved from the symbol table.

The return address in F_n's stack frame is an address within the previously invoked routine F_{n-1}. The debugger uses that address to retrieve the parameter list and name of F_{n-1} from the symbol table. The stored frame pointer in F_n's stack frame points to F_{n-1}'s stack frame. The values of the arguments to F_{n-1} are retrieved by the debugger through this stored frame pointer, formatted appropriately, and displayed. The debugger repeats this process, using information found in the stack frame of the just-displayed routine to display that routine's caller until all routines have been displayed. We will call this the $Fchain$ method.

An algorithm to construct a call-stack trace using the $Fchain$ method is given in Figure 2.4.

3 The Problem

If one or more of the frame pointers have been optimized away, a debugger using the $Fchain$ method will construct a call-stack trace that is incorrect. If a single frame pointer (for routine F_i) has been optimized away, the debugger will construct a call-stack trace that associates F_i's name with the stack frame of F_{i-1} (formatting the values found there according to the symbol table entry for F_i), and neither F_{i-1}'s name nor F_i's arguments will appear. If the frame pointers for routines F_i through F_{i+j} have been optimized away, the debugger will construct a call-stack trace that associates F_{i+j}'s name with the stack frame of F_{i-1} and no information about F_i through F_{i+j-1} will appear. Figure 2.3 shows a call stack containing four active routines, one of which has had this optimization performed on it. The call-stack trace produced by the $Fchain$ method on this call stack is:

F_3's name(F_3's arguments)
F_2's name(garbage)
F_0's name(F_0's arguments)

Although four routines are active, the call-stack trace contains only three entries, one of which is incorrect.

4 Solutions

The general approach to the problem is to have the debugger use an alternative method of constructing the call-stack trace that does not rely on the frame pointers in the call stack. Several solutions are presented.

[4] A debugger saves the values that are in the machine registers when it takes control at a breakpoint, thus when we use the terms "instruction pointer", "frame pointer register", and "stack pointer register" we are actually referring to the debugger's copy of the values that were in these registers when the breakpoint was reached.

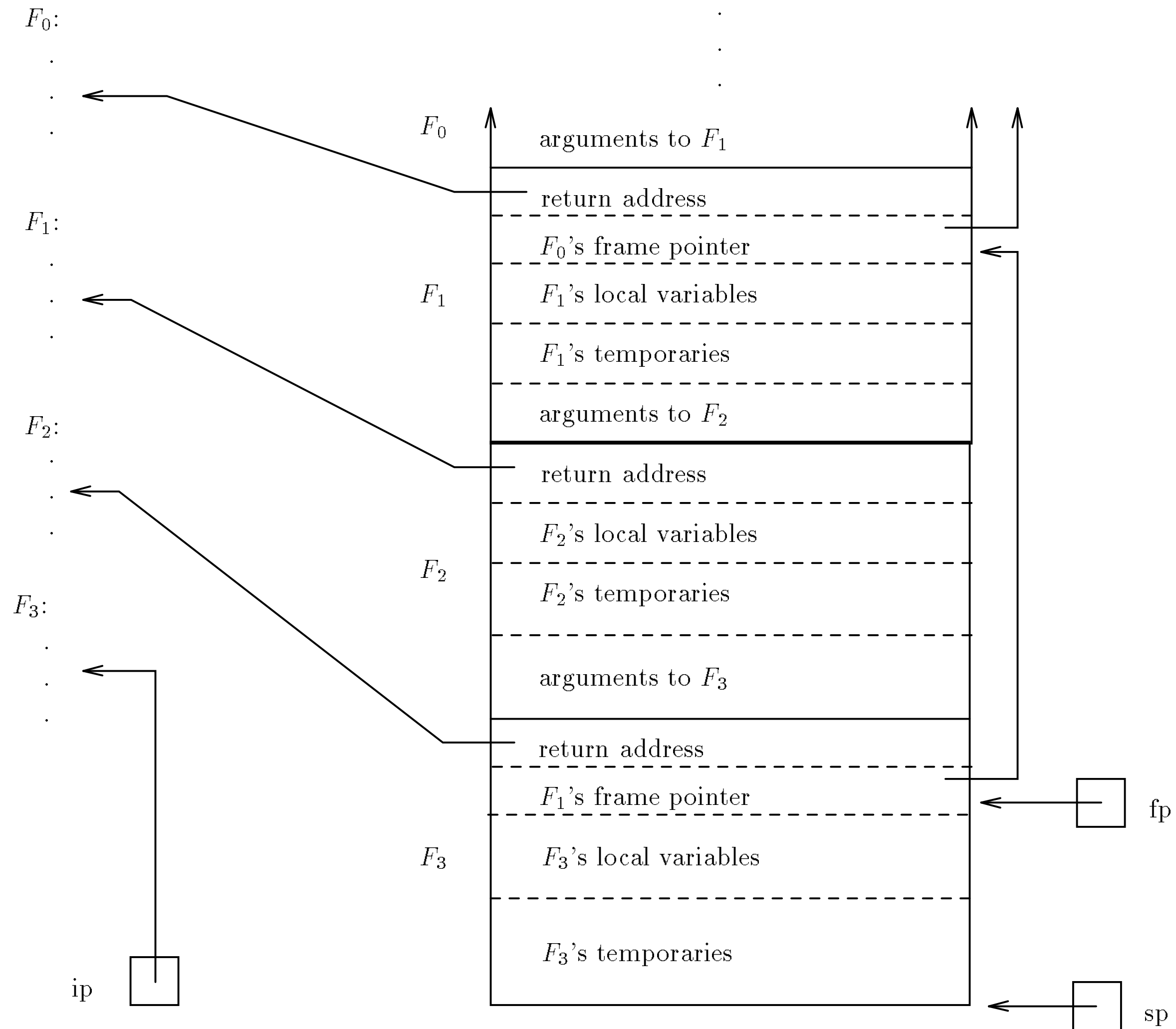


Figure 2.3: Optimized Call-Stack

4.1 The Debugger Maintains Its Own Frame Pointers

The debugger can maintain its own copy of the information that it currently gets from the call stack. The debugger uses *invisible breakpoints* ([Zel84]) to collect the information. A invisible breakpoint is a breakpoint at which the debugger halts the executing program, takes some action, and continues execution without ever giving control to the user.

As we have seen, in order to construct the call-stack trace, the debugger needs for each routine F:

- an address of some instruction within F, which it uses to locate the symbol table entry for F, and
- a pointer to the base of F's stack frame, where it finds F's arguments.

Recall that the first instruction in a routine's prologue pushes the caller's frame pointer into the current stack frame (following the return address). Once that push has occurred, the stack pointer register points to the location that by convention is considered the base of the routine's stack frame. If the debugger were to take control immediately after the first instruction in the prologue, it could make a copy of the value in the stack pointer register, which would give it a pointer to the base of the stack frame, and it could make a

In the following algorithm, *ip* is the instruction pointer register, *fp* is the frame pointer register, and we assume the debugger has the following routines available to it:

- *get-symbol-table-information*, which takes a code address and determines which symbol table entry corresponds to the routine containing that address, then returns the name and parameter type list from the symbol table entry,
- *display-routine-entry*, which takes a routine name, a parameter type list, and a frame pointer, and finds the arguments in the stack frame pointed to by the frame pointer, formats them according to the type list, and displays the routine name and appropriately formatted arguments,
- *get-return-address*, which takes a frame pointer and returns the return address that is stored in the stack frame pointed to by that frame pointer, and
- *get-frame-pointer*, which takes a frame pointer and returns the frame pointer that is stored in the stack frame pointed to by that frame pointer.

The termination condition given here is somewhat arbitrarily when the "main" routine (the entry point of the user's code) has been displayed. Actual termination conditions are system dependent.

Algorithm *Fchain*-**Call-Trace:**

```
routine-address ← ip
frame-pointer ← fp
repeat
    name, parameter-types ← get-symbol-table-information(routine-address)
    display-routine-entry(name, parameter-types, frame-pointer)
    routine-address ← get-return-address(frame-pointer)
    frame-pointer ← get-frame-pointer(frame-pointer)
  until name = "main"
```

Figure 2.4: Debugger Algorithm to Display a Call-Stack Trace using the *Fchain* Method

copy of the value in the instruction pointer register, which would give it an address of an instruction within the routine. By setting an invisible breakpoint at the second instruction in each routine, the debugger can get the information that it needs for a call-stack trace.

Note that if the debugger set its invisible breakpoint at the *first* instruction in the prologue, it can still get the information that it needs for the call-stack trace. As in the above case, the instruction pointer contains an address within the routine. The stack pointer register contains at this point a value that must be offset by the size of the frame pointer that is about to be pushed in order to get a pointer to the base of the stack frame.

This scenario assumes an unoptimized stack frame. We are interested in the optimized case, when the push of the caller's frame pointer into the stack frame does not occur. However, we have just seen that the debugger can get the information it needs by setting its invisible breakpoint at the first instruction in the prologue. Since the debugger can get this information before the instruction executes, it clearly doesn't matter whether that instruction saves the caller's frame pointer.

The debugger must set a breakpoint at the first instruction of every routine. Each time a routine is called, the debugger copies two pieces of information into its own workspace. It uses this information rather than the information that may (not) be stored in the call stack to construct the call-stack trace.

Clearly, for this to be correct, the debugger must also set an invisible breakpoint at the return instruction of every routine so that it can remove the information for the about-to-return routine from its workspace –

otherwise, it would be maintaining not a call-stack trace, but a subroutine-call history. That is, the debugger must store this information in a stack of its own, pushing when a routine is called and popping when the routine returns. We call the debugger's stack the *dstack*. Assuming the debugger correctly maintains its dstack, it can use the information therein just as it would have used the corresponding information that it would find in the program's call stack, to examine the symbol table and to access parameters and local variables. We will call this the *Dstack* method.

The locations at which invisible breakpoints must be set are call and return instructions. The locations of the call instructions are available from the symbol table. Either additional compiler support is needed to ensure that the locations of the return instructions are also placed in the symbol table or the debugger must scan the executable and locate them before executing the program.

4.2 A Cheaper Method: The Debugger Maintains Only the Missing Frame Pointers

Subroutine calls are extremely common. The *Dstack* method has a considerable amount of overhead with two invisible breakpoints per call – probably an untenable amount of overhead. The *Dstack* method and the *Fchain* method can be combined to bring the overhead down to an acceptable level. The dstack is entirely redundant if the frame pointer is set up by every routine. It is partially redundant if the frame pointer is optimized away by some routines. Redundancy can be limited by only creating entries in the dstack for routines that do not set up a frame pointer. The compiler can tell the debugger which routines set up frame pointers. The symbol table entry for a routine must be extended to include a field with a boolean entry recording the presence or absence of a frame pointer. There is now overhead of two invisible breakpoints per call only for those routines that do not use a frame pointer.

Even so, the dstack remains partially redundant. It contains only frame-pointers that are not present in the call stack. But each dstack entry also contains an address. The address in the entry for F_i is redundant with the return address which has been placed in the stack frame of F_{i+1} by the call instruction. The dstack can therefore be simplified to contain only frame pointers. We will call this the *Dstack/Fchain* method.

Consider how the debugger will construct the call-stack trace given an optimized call stack and a dstack. The basic difference is where a pointer to the stack frame of each routine is found. Note that the frame pointers stored in the call stack still form a linked list, but now they chain only stack frames of routines that use frame pointers. If F_i uses a frame pointer (indicated by F_i's symbol table entry), then either the frame pointer register or the frame pointer in the stack frame of subsequently called active routine F_{i+j} points to F_i's stack frame, where F_{i+j} is the next routine that uses a frame pointer. If F_i does not use a frame pointer, then the dstack contains a pointer to its stack frame.

An algorithm to construct a call-stack trace for an optimized call stack using the *Dstack/Fchain* method is given in Figure 4.1.

4.3 Non-Local Gotos

Commonly used procedural languages provide some form of jump from the middle of a routine to the middle of some other previously invoked active routine, a jump that is not simply a return to the calling routine. Sometimes this is provided in the language as part of the **goto** facility (as in Pascal), and it is from there that the name non-local goto was coined – it is a **goto** whose target is not local to the routine that the **goto** is in. In other languages, such as C, library routines perform the same function (in C, these routines are **setjmp** and **longjmp**). Other languages provide some form of exception handling, which can have the same effect.

Any of these forms of non-local goto cause difficulties for the *Dstack* and *Dstack/Fchain* methods. In order to maintain correctness in its dstack, the debugger must reach invisible breakpoints at both the beginning and the end of each routine that does not set up a frame pointer. If a non-local goto occurs, the end of such an active routine may never be encountered, since the routine does not perform a normal return.

```
Algorithm Dstack/Fchain-Call-Trace:
    routine-address ← ip
    call-stack-frame-pointer ← fp
    temp-dstack ← dstack // copy dstack so pops are not destructive
    dstack-frame-pointer ← pop(temp-dstack)
    repeat
        name, parameter-types ← get-symbol-table-information(routine-address)
        if (uses-frame-pointer(routine-address))
            frame-pointer ← call-stack-frame-pointer
            call-stack-frame-pointer ← get-frame-pointer(frame-pointer)
        else
            frame-pointer ← dstack-frame-pointer
            dstack-frame-pointer ← pop(temp-dstack)
        display-routine-entry(name, parameter-types, frame-pointer)
        routine-address ← get-return-address(frame-pointer)
      until name = "main"
```

Figure 4.1: Algorithm to Display a Call-Stack Trace in the Occasional Absence of Frame Pointers using the *Dstack/Fchain* Method

Instead, the return is performed for it by restoring the machine to an earlier saved state. There is a problem only if a routine F_i that does not set up a frame pointer has been called and has not returned, and the target of the goto is code in a routine F_{i-j} called earlier than F_i. F_i will not return normally, the invisible breakpoint at F_i's return statement will not be reached, and the debugger will not pop its entry for F_i from the dstack. Let us assume there is a routine F_{i-j-k} that was called earlier than F_{i-j} (thus is active after the non-local goto) and F_{i-j-k} does not set up a frame pointer. After the non-local goto, the debugger is asked for a call-stack trace. An example of such a situation is shown in Figure 4.2. The debugger algorithm displays the call-stack trace entries correctly until it attempts to display F_{i-j-k}. It determines from the symbol table entry for F_{i-j-k} that F_{i-j-k} has not set up a frame pointer, so it uses the entry on the top of dstack to continue the construction of the call-stack trace. This, of course, is the wrong entry – it is the entry for F_i. Two solutions to this problem are given below.

One Solution: Cleaning Up the Dstack

If immediately after a non-local goto was executed the debugger took control, it could remove any inappropriate entries from the top of the dstack. Any entry with a frame pointer pointing higher on the stack than the value in the stack-pointer register is an entry that should be removed, since such entries must be for routines that have "returned" via the non-local goto.[5] The debugger must take this action right away, since subsequent subroutine calls will cause the stack to grow.

For languages like C, in which non-local gotos are implemented by library routines, the debugger can set an invisible breakpoint at the library routine that implements the goto, and when it reaches such a breakpoint, the debugger can clean up the dstack. This may work for languages with exception handling as well.

Languages with direct jumps in the code require some work on the part of the compiler if the invisible breakpoint cleanup solution is to be used. The debugger must be told where to set its invisible breakpoints, so the compiler must provide a list of such addresses to the debugger.

[5] On many architectures the stack grows down, so higher on the stack may mean a lesser value in the frame pointer.

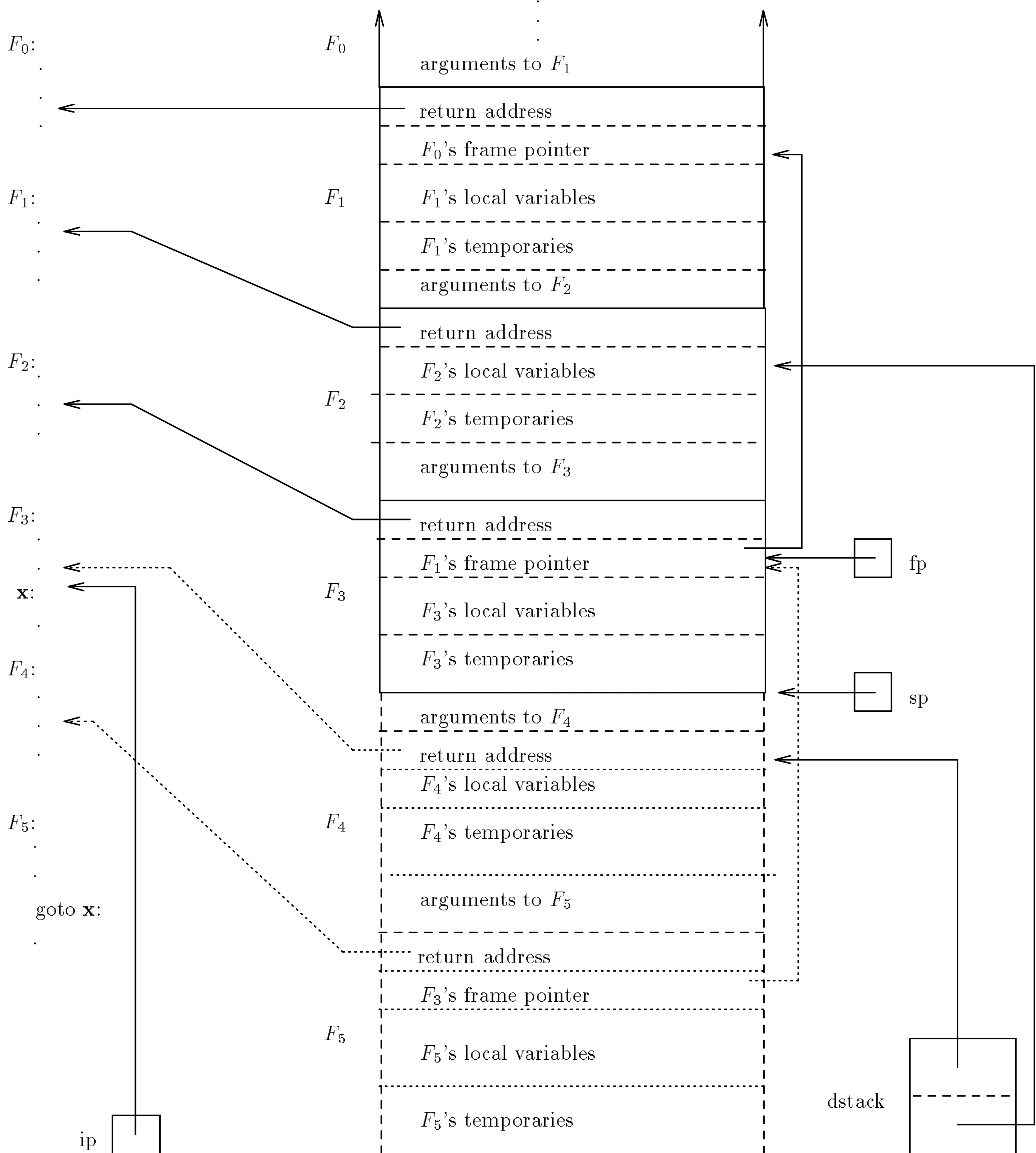


Figure 4.2: Stackframe After Non-Local Goto

Another Solution: Leaving Redundancy in the Dstack

In the *Dstack* method, because the dstack alone is used to construct call-stack traces, each entry must contain both a frame pointer and a code address. We noted above that when both the call stack and the dstack are used, the code addresses in the dstack are redundant. This redundancy can be exploited.

In constructing a call-stack trace, when the debugger needs to use a dstack entry, it already has a code address A within the routine that is to be displayed next (the code address that was the return address found in the stack frame of the routine that was previously displayed). If the code address in the current dstack entry and A are not addresses within the same routine, the debugger can simply ignore the dstack entry and go on to the next entry. Whenever the invisible breakpoint for the return of a routine that does not use a frame pointer is reached, all dstack entries must be popped until (and including) the entry whose code address is in the same routine as the return instruction.[6]

This scheme does not require that the debugger take control immediately after a non-local goto, and thus does not require any special effort on the part of the compiler. It does increase the expense of constructing a call-stack trace, since more code addresses must be matched with the routines that contain them.

4.4 A "Free" Method: The Compiler Does the Work

A routine may contain numerous calls to other routines. Assume routine A calls routines B and C. The size of A's stack frame at the point of the call to B may differ from the size of its stack frame at the point of the call to C. However, the size of A's stack frame at the point of the call to B may be the same every time A is active. If the size of a routine's frame is constant at the point that it makes a call, the size can be determined by the compiler. The return address stored in the stack frame of the called routine is a unique identifier of the call. A table *Fsize* of </em return address, *size of caller's stack frame at point of call*> pairs can be provided by a compiler. We can use such a table to compute the appropriate position for each frame pointer without looking in the run-time stack (thus optimization of the calling sequence does no harm). Assuming a lookup function *get-frame-size* for this table that takes a return address and returns the associated stack frame size, and function *get-return-address* as defined in Figure 2.4, the position FP_i of the (logical) frame pointer for the ith function on the stack can be determined by:

$$FP_i = FP_{i+1} - get{-}frame{-}size(get{-}return{-}address(FP_{i+1}) \tag{4.1}$$

The algorithm given in Figure 2.4 can be used to display the call-stack trace if function *get-frame-pointer* is redefined to return FP_i as defined in equation 4.1.[7] No invisible breakpoints need to be set by the debugger. We call this the *Fsize* method. The *Fsize* method is elegant but incomplete:

1. it fails for routines whose stack-frame size is not constant at the point of a call made by the routine, and
2. it fails if the return address of an active routine is not available. Once a frame pointer is available, the return address of all previously invoked routines can be found. However, if the routine that was executing when the debugger gained control (hereafter the *current* routine) does not set up a frame pointer, the frame-pointer register cannot be used to find the return address of its caller. That return address is present on the stack, but its location is not exactly identified.

---

[6] If an invalid entry is on the top of the dstack when a call-stack trace is being constructed, it can be popped from the dstack then. However, an entry may be buried on the dstack because subsequent to the non-local goto other routines were called.

[7] The problem would be much simpler if the size of A's stack frame were identical for all calls made by A. The stack-frame size could be included in A's symbol table entry, eliminating the need for *Fsize*. Unfortunately, if the stack pointer is used to reference temporaries, fixing the stack frame size would require modification of the stack pointer prior to and following calls, which would cost more than is saved by optimizing the calling sequence. However, in some architectures, such as Mips, the stack pointer is not used in this manner, and this approach is taken [Cor91].

```
Algorithm Fsize/Fchain-Call-Trace:
    routine-address ← ip
    call-stack-frame-pointer ← fp
    repeat
        name, parameter-types ← get-symbol-table-information(routine-address)
        if (uses-frame-pointer(routine-address))
            frame-pointer ← call-stack-frame-pointer
            call-stack-frame-pointer ← get-frame-pointer(frame-pointer)
        else
            frame-pointer ← frame-pointer - get-frame-size(get-return-address(frame-pointer))
        display-routine-entry(name, parameter-types, frame-pointer)
        routine-address ← get-return-address(frame-pointer)
      until name = "main"
```

Figure 4.3: Algorithm to Display a Call-Stack Trace in the Occasional Absence of Frame Pointers using the *Fsize/Fchain* Method

When the Frame Size Varies

If the size of a routine's stack frame at the point of a particular call is not constant, the caller's frame pointer must be saved, that is, the prologue and epilogue of the called function must not be optimized. The chain of frame pointers stored in the stack can be used to locate stack frames of such routines while the *Fsize* table can be used to locate stack frames of routines that do not save the frame-pointer register. We call this the *Fsize/Fchain* method, and it is the solution that we implemented.

An algorithm to construct a call-stack trace for an optimized call stack using the *Fsize/Fchain* method is given in Figure 4.3.

When the Current Routine Has No Frame Pointer

If the current routine does not set up a frame pointer, the location of the return address of its caller is not exactly identified. One option is to search the stack for the return address. The search is bounded on one side by the value in the stack pointer register and on the other by the value in the frame pointer register. The stack search may be complex (it may have to be done more than once, starting at different alignments), but the return address is guaranteed to be on the stack. There is a chance of finding a value in the stack that matches an *Fsize* return-address entry but is not the caller's return address. This chance is probably fairly small (unless the program is a compiler or debugger), but is nonzero. The probability of such an error can be decreased by augmenting *Fsize* with a third field containing the name of the called function.

Fsize would then consist of <*caller's return address, caller's framesize, callee's name*> triples. If a value that matches the first field is found in the stack, the current function's name can be tested against *Fsize*'s third field. We call this the *Fsize+/Fchain* method. It is still not foolproof: assume A is the current function, called by B with return address R, and also somewhere in the code there is a call of A by C with return address P. If a value equal to P happens to be on the relevant portion of the stack above R, the wrong *Fsize* entry will be used. However, this is considerably less likely than the previous error probability.

The callee's name may not be known (the call may be through an address passed in to the caller). For such calls, the third field in *Fsize* may be left null. A null *Fsize* entry could be considered to match any routine name. Alternatively, the compiler could put out a table T of routines whose addresses have been taken; a null *Fsize* entry could be considered to match only entries in T. This opens the door to error in

	Total Compilation Time	Sum of Object Module Sizes
Fchain	804.3 seconds	490987
Fsize/Fchain	825.4 seconds	589113
increase	2.5 per cent	20 per cent

Table 5.1: Compilation times and object module sizes for the *Fchain* and *Fsize/Fchain* methods. A compiler benchmark consisting of 44 source files containing a total of 13,511 lines of C code was compiled.

the other direction, where the actual return address is rejected because it is a call of a routine that is not in T.[8]

A simpler alternative is to just provide truthful behavior if the active function has an optimized prologue: give its name, state that the stack trace may be missing some functions, and start the trace at the function whose frame is pointed to by the frame-pointer register.

5 Implementation

We have a prototype implementation of the *Fsize/Fchain* method using MetaWare Incorporated's High C compiler and The Free Software Foundation's gdb debugger, running on an MC68000 based Sun workstation.

Compiler modifications involved adding approximately 50 lines of code spread across seven source files, none of it inside loops. The 'pushes fp' bit was integrated into the existing symbol table format at no space cost. *Fsize* takes eight bytes per call (four bytes for the return address, four bytes for the frame size) plus four bytes per file (to record the number of entries). We used compiler benchmarks to compare compilation times and object module sizes for the *Fchain* and *Fsize/Fchain* methods. Table 5.1 gives the time to compile one benchmark program consisting of 13,511 lines of C code (in 44 source files), using each method. The table also gives the sum of the sizes of the object modules produced. For both compilation time and object module size, the increase due to the *Fsize/Fchain* method is given as a percentage of the time or size associated with the *Fchain* method. Individual object-module size increases varied from less than one per cent to 34 per cent. The object module size increases would be larger for the *Fsize+/Fchain* method. No attempt was made to optimize for space.[9]

Debugger modifications involved adding approximately 150 lines of code spread across five source files. The time the modified debugger spent on call-stack traces was not perceptibly different from the time spent by an unmodified debugger.

6 Solution Summary and Comparison

The enabling technology for producing an accurate call-stack trace in the occasional absence of frame pointers is either the *dstack* or *Fsize*. In this section we summarize and compare the *Dstack/Fchain* and *Fsize+/Fchain* methods. First we break the methods down by what is required of the compiler and debugger. We include the effect on the symbol table format, because it is often the limiting factor in what information is passed from the compiler to the debugger.

[8] For example, on some systems interrupt handlers are placed in constant locations. These routines would not be in T.

[9] There is considerable potential for space optimization in *Fsize*, at the cost of added complexity in reading the table. The return address could be entered as an offset from the function entry point. Typical stack frame sizes can be expressed in a few bits rather than four bytes. If *Fsize* were optimized for space, the *Fsize+* version with the caller's name could probably be implemented in half the space of our existing implementation that does not include the caller's name.

- *Dstack/Fchain*
  The debugger must be able to determine where the first instruction in each routine is located, and where the routine exits (return instructions) are located. The debugger could determine this information by scanning the executable code. This would slow debugger start-up, but would allow this method to be used with no compiler support. We assume, however, that the task of producing this information would fall to the compiler. In addition, the debugger must be able to determine whether a routine's prologue pushes a copy of the frame pointer register. As before, while the debugger could determine this information by scanning the executable code, we assume that the compiler will be responsible for supplying the information to the debugger.
  - Symbol Table Format
    The symbol table already has a place for the location of the first instruction in a routine. It must be augmented to include the routine exit locations, and to include a 'pushes fp' bit encoding whether a routine's prologue pushes a copy of the frame pointer register.
  - Compiler
    The compiler must set the 'pushes fp' bit and record each routine exit location in the augmented symbol table.
  - Debugger
    The debugger must read the augmented symbol table and place invisible breakpoints at the entry and exit of each routine whose 'pushes fp' bit is off, prior to running the target program. On reaching an invisible breakpoint at routine entry, the debugger must push the stack pointer onto the dstack; on reaching an invisible breakpoint at routine exit, the debugger must pop the dstack. The debugger must use dstack entries to locate frames of these routines, and use frame pointers saved in the call stack to locate frames of other routines.
- *Fsize+/Fchain*
  - Symbol Table Format
    The symbol table must be augmented to include the *Fsize* table of <*return address*, *framesize*, *name*> triples. It must also include the 'pushes fp' bit in the entry for a routine, as above.
  - Compiler
    For each call, the compiler must record the caller's framesize at the point of the call instruction along with the address of the following instruction and the name of the called routine in *Fsize*. For each routine, the compiler must set the 'pushes fp' bit in the routine's symbol table entry.
  - Debugger
    The debugger must use *Fsize* entries instead of frame pointers saved in the call stack to locate frames of routines whose 'pushes fp' bit is off. The debugger must use frame pointers saved in the call stack to locate frames of other routines. Because *Fsize* is indexed by return addresses, if the current routine's 'pushes fp' bit is off, the debugger must either
    1. print its name only, warn the user that the stack trace may be incomplete, and provide a stack trace beginning with the routine whose frame is pointed to by the frame-pointer register, or
    2. search the stack for a return address that appears in the first field of some *Fsize* entry whose third field is the name of the current routine; the debugger must use that *Fsize* entry to locate the current routine's stack frame.

Next we summarize the advantages and disadvantages of these method relative to each other.

- *Dstack/Fchain*
  - Disadvantages
    This method requires additional complexity in the debugger. When invisible breakpoints are set, the *Dstack* method has an overhead of hundreds of instructions per call of routines that do not save frame pointers in their stack frame. When invisible breakpoints are not set, the *Dstack* method has no run-time overhead. The symbol table entry for every routine must be read, and the invisible breakpoints set, before the *Dstack* method can be used. This can cause a significant

delay in debugger start-up. The location of routine exits must be available to the debugger. This either adds to the complexity of the symbol table and the compiler and to the size of the object modules and executable (typically 4 bytes per routine for the routine exit and one 'pushes fp' bit per routine) or adds to the start-up time and complexity of the debugger.

  - Advantages
    The *Dstack/Fchain* method can be implemented in a debugger without a dependence on compiler support. It can always provide a correct call-stack trace.

- *Fsize+/Fchain*
  - Disadvantages
    This method requires additional complexity in the symbol table, the compiler, and the debugger. The size of the object modules and executable increases significantly. Compilation time increases a little, as does the time required to read the symbol table. In the case that the current routine does not save the frame-pointer register, the method is either incomplete or may be incorrect.
  - Advantages
    The time cost to the debugger of a larger symbol table is more than offset by the ability to read the table on demand: unlike the *Dstack/Fchain* method, the *Fsize/Fchain* method only requires the symbol table entry for a routine to be read if the routine is active when the call-stack trace is performed. The method has no run-time overhead.

7 Conclusion

The method used today by interactive debuggers to provide a call-stack trace relies on a frame pointer being set up within the stack frame of each active subroutine. This paper gives several methods that support a debugger's call-stack trace facility in the circumstance that, due to optimization, some routines do not set up a frame pointer. These methods vary in their costs: there is a trade-off between run-time overhead for the debugger and required symbol table and compiler support.

References

[CM91] M. Copperman, C. E. McDowell, "Debugging Optimized Code Without Surprises," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[Cop90] M. Copperman, "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values," University of California, Santa Cruz technical report UCSC-CRL-90-23, May 1990.

[Cor91] Steve Correll, personal communication, Borland International, Scotts Valley, CA, April 1991

[CMR88] D. Coutant, S. Meloy, M. Ruscetta "DOC: a Practical Approach to Source-Level Debugging of Globally Optimized Code," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[FM80] P. H. Feiler, R. Medina-Mora, "An Incremental Programming Environment," Carnegie Mellon University Computer Science Department Report, April 1980.

[Hen82] J. Hennessy, "Symbolic Debugging of Optimized Code," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, 1982.

[PS88] L. L. Pollack, M. L. Soffa, "High Level Debugging with the Aid of an Incremental Optimizer," *Hawaii International Conference on System Sciences*, January 1988.

[PS92] L. L. Pollack, M. L. Soffa, "Incremental Global Reoptimization of Programs," Draft from Department of Computer Science, University of Pittsburgh, May 1991. To appear in *ACM Transactions on Programming Languages and Systems* in 1992.

[Str91] L. Streepy, "CXdb A New View On Optimization," *Proceedings of the Supercomputer Debugging Workshop* , Albuquerque, November 1991.

[WS78] H. S. Warren, Jr., H. P. Schlaeppi, "Design of the FDS interactive debugging system," IBM Research Report RC7214, IBM Yorktown Heights, July 1978.

[Ze83a] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," *Research Report CSL-83-1* , Xerox Palo Alto Research Center, Palo Alto, CA, Jan. 1983.

[Ze83b] P. Zellweger, "An Interactive High-Level Debugger for Control-Flow Optimized Programs," *SIGPLAN Notices*, Vol. 18, No. 8, pp. 159-172 Aug. 1983.

[Zel84] P. Zellweger, "Interactive Source-Level Debugging of Optimized Programs," Research Report CSL-84-5, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

[ZJ90] L. W. Zurawski, R. E. Johnson, "Debugging Optimized Code With Expected Behavior," Unpublished draft from University of Illinois at Urbana-Champaign Department of Computer Science, August 1990.