Methods for Saving and Restoring Register Values across Function Calls

JACK W. DAVIDSON AND DAVID B. WHALLEY
Department of Computer Science, University of Virginia, Charlottesville, VA 22903, U.S.A.

SUMMARY

The method used to save and restore the values of registers across function calls can affect performance and influence the design of specific instructions. This paper describes the results of an experiment that empirically evaluated six different schemes for saving and restoring registers on CISC machines. The methods do not require special hardware or interprocedural analysis to be performed. The six schemes are logically divided into two classes. Three of the techniques do not require the compiler to perform data-flow analysis while the other three schemes do. Within each class one scheme delegates the responsibility of preserving the state of the registers to the calling function. The second scheme of each class delegates the responsibility of saving the registers to the function being called. In the third scheme of the two classes, the registers are partitioned into two disjoint sets. The calling function is responsible for preserving register values in one of the sets while the called function is responsible for the other set. For each class the third scheme is shown to produce the most effective code.

KEY WORDS: Calling sequences Registers Save/Restore Code Generation Calling Conventions

INTRODUCTION

There is a sequence of actions that is required to implement function calls. This sequence of events, or ‘‘calling convention’’, is an important aspect of both machine design and programming language design and implementation. Poor architectural support for subprogram calls or poor design of the calling convention can result in slower programs and may lead programmers to avoid the subprogram abstraction mechanism.

An important component of a calling convention is how to preserve the values of the registers that may be used across function calls. Without link-time optimizations[1] , interprocedural analysis [2], or special hardware, such as register windows[3] or dynamic masks[4] , there are two methods commonly used to save and restore the values in registers that may be destroyed by the called function. One method delegates the responsibility for saving and restoring the values of the registers to the called function. Upon entry to the called function, the values of the registers that are used in that function and need to be preserved are stored. The values of these registers are then loaded upon exiting the called function. For languages that support recursion, the register values are usually saved on and restored from the activation record of the called function. Since the responsibility of saving and restoring the register values is

delegated to the called or callee function, this method is known as ‘‘callee-save’’.

The other method delegates the responsibility for saving and restoring the registers to the calling function. The calling function stores the values of the registers on the run-time stack before a call instruction. After the call instruction, the function loads the values from the run-time stack into the registers. Since the responsibility of saving and restoring the register values is assigned to the calling or caller function, this method is known as ‘‘caller-save’’.

There are many factors to consider when choosing a method to save and restore the values in registers across function calls. Choosing a method often involves trading off implementation simplicity for run-time efficiency. To a large extent, design decisions are driven by the architectural support supplied by the target machine and the intuition (bias) of the implementor. This paper describes the results of a set of controlled experiments that were used to evaluate several methods for saving and restoring registers on CISC machines. Our experiments show that a hybrid approach, a combination of callee and caller methods, produces the most effective code.

RELATED WORK

There has been some previous work studying methods for saving and restoring registers. In seeking to show that their register window mechanism would efficiently support high-level languages, Patterson and Sequin measured how deeply calls are nested at run-time[3] . They found that the average calling depth during the execution of a program was not very great. Their hardware approach, register windows, avoids saving and restoring registers by switching to a new window at each call. Saves and restores are required only when no more windows are available. With eight register windows, less than one percent of the calls and returns required a window to be stored and loaded from memory.

Lang and Huguet[4] analyzed the effectiveness of a simple caller-save method, a simple callee-save method, and six other schemes involving a dynamic mask. Each bit set in a single dynamic mask indicates a register whose value needs to be retained by functions currently active. A static mask, associated with each function, indicates the set of registers it uses. The set of registers saved and restored is computed by taking the intersection of the dynamic and static masks. For each of these schemes, variables

were preallocated to registers. They found that the dynamic mask could reduce the total number of saves and restores. They also found that the dynamic mask schemes benefited from a larger set of registers by assigning registers in a round-robin fashion as functions were being compiled.

Chow[2] investigated linking program units together in a compilation environment to allow interprocedural register allocation. He divided the registers of a machine into caller-save and callee-save sets. Using a technique called shrink-wrapping, the saves and restores for callee-saved registers were placed only around regions where the registers were used. By processing the functions in a depth-first ordering of the call graph, the interprocedural register allocator avoided using registers that could be active at the same time in other functions. It also was used to pass arguments through registers. With a sufficient number of registers, Chow found that the cost of saving and restoring registers at procedure calls could be significantly reduced.

ENVIRONMENT FOR EXPERIMENTATION

To investigate the effect of employing different methods for saving and restoring registers across function calls, we used an environment called *ease* (Environment for Architecture Study and Experimentation) [5,6]. *ease* consists of a compiler generation system and measurement tools. The compiler generation system, *vpo*[7] , permits compilers to be constructed easily and quickly for a new architecture. It is retargeted by supplying a description of the target machine's architecture. The compiler also supports trying out different code generation strategies. That feature is used to implement the different register save and restore conventions. The compiler generates production-quality code. Indeed, the C compiler is being used commercially and the back end was used to construct a commercial Ada compiler.

The measurement tools permit very detailed analysis of the execution behavior of programs. For example, *ease* can gather statistics about how many instructions were executed, the number of memory references (reads and writes) performed, as well as a complete breakdown of which instructions and addressing modes were used. The technique it uses adds very little overhead to a program. A program being measured runs approximately 10 to 15 percent slower than a non-instrumented program. This permits real programs to be used for gathering measurements.

For these experiments, we measured the number of instructions executed, the number of memory references, and the size of the object code. The experiments were performed on a VAX-11 and a Motorola 68020. On both of these CISC machines, C compilers were constructed for each of the six methods presented in the following section. The only calling convention changes made to the run-time library were those required to implement the new save and restore methods. Other conventions, such as passing arguments on the stack instead of in registers, were not altered.

The number of instructions executed is affected by two factors. Typically, as more variables are allocated to registers, the number of instructions used for saving and restoring registers increases. On the other hand, as frequently used variables are allocated to registers, the number of instructions aside from those used for saving and restoring registers decreases. This occurs since *vpo* reattempts code selection after register allocation and often more optimizations are possible once a variable has been allocated to a register. For instance, the VAX-11 and Motorola 68020 autoincrement and autodecrement addressing modes are typically used only if a variable is allocated to a register. Use of these addressing modes results in fewer instructions being executed.

The number of dynamic memory references is also affected by the number of variables allocated to registers. As more variables are allocated to registers, the number of memory references for saving and restoring registers increases. Conversely, the number of other types of memory references, loading and storing values associated with references to variables, decreases.

The test set used for our experiments consisted of nineteen programs distributed over forty-five files (more than 22,000 lines of C code). On the VAX-11, all source files of each program were recompiled. To more accurately determine the impact of each method, the source files from the C run-time library were also recompiled. The test set comprised a total of 250 source files (including files from the C library). The C library on the SUN-3, a Motorola 68020-based machine, was not used in data collection since the sources for the library routines were not available. Thus, the test set used for the Motorola 68020 was the same nineteen programs consisting of only forty-five source files. Data was collected from each of the files compiled by *vpo*. Since the C library was not used for data collection on the Motorola 68020, the results for the VAX-11 more accurately reflect the effectiveness of each method of

saving and restoring registers. The set of test programs is summarized in Table 1.

Table 1: Test Set

Class	Name	Description or Emphasis
Unix System Utilities	cal	Calendar Generator
	cb	C Program Beautifier
	compact	Huffman Coding File Compression
	diff	Differences between Files
	grep	Search for Pattern
	nroff	Text Formatting Utility
	od	Octal Dump
	sed	Stream Editor
	sort	Sort or Merge Files
	spline	Interpolate Smooth Curve
	tr	Translate Characters
	wc	Word Count
Benchmark Programs	dhrystone	Synthetic Benchmark Program
	matmult	Multidimensional Arrays and Simple Arithmetic
	puzzle	Recursion and Array Indexing
	sieve	Simple Iteration and Boolean Arrays
	whetstone	Arithmetic Operations
User Code	mincost	VLSI Circuit Partitioning
	vpcc	Very Portable C Compiler

METHODS INVESTIGATED

The technique used to save and restore register values across function calls can affect performance and influence the design of the instruction set of a machine. The advantages and disadvantages of the possible implementations are not obvious. To better understand the tradeoffs, six possible implementations for saving and restoring register values across function calls were examined. These six methods are categorized in Table 2.

Table 2: Save/Restore Methods Investigated

No data flow information available	Data flow information available
Simple callee	Smarter callee
Simple caller	Smarter caller
Simple hybrid	Smarter hybrid

Examination of the results from experiments with these methods can provide insight for answering the following questions.

(1) What is the most effective method for minimizing the number of saves and restores in a simple compiler that does not perform data-flow analysis?

(2) If the compiler performs data-flow analysis, which approach for saving and restoring registers results in the best performance and what is the performance improvement over approaches that do not require data-flow information?

For each method, *vpo* attempts to allocate local variables and arguments to registers if the expected benefits from doing so outweigh the save/restore cost. The benefits are determined by estimating the number of times that a variable will be referenced when the function is invoked. Typical of many compilers, references inside loops are more heavily weighted. Currently, *vpo* does not use any of the popular graph coloring techniques for register allocation[8,9] which can often reduce the number of registers used by a function. While the choice of a register allocation algorithm can change the values of the measurements obtained, the relative merits of the register save/restore methods should be unaffected. The following subsections describe the attributes of the six methods investigated and their implementation using *vpo*.

Simple Callee

Because of its simplicity, the simple callee method for saving and restoring registers is widely used. With this approach, the set of allocable registers is broken into two groups. One group, called *non-scratch*, is used to hold local variables and arguments. The other group, *scratch* registers, are not guaranteed to retain their values across function calls. Consequently, they are only used to hold temporary values, to compute intermediate results, and to return values from the called function to the calling function. Only the non-scratch registers that are used in the function are saved and restored upon function entry and exit respectively.

The simple callee method is used by *pcc*-based C compilers [10] on the VAX-11 running 4.3BSD Unix and on the SUN-3 running SunOS 4.0. For both of these compilers, a register declaration of a local

variable or argument results in that variable being allocated to a non-scratch register if one is available. On the VAX-11, a mask with a bit set for each register to be preserved is stored at the beginning of each function. The `calls` instruction scans the mask pushing the contents of the indicated registers onto the run-time stack. On function exit, the `ret` instruction restores the values into the appropriate registers. On the SUN-3, a Motorola 68020-based machine, special instructions are used to save and restore the non-scratch registers referenced by the called function. These instructions also use a bit mask to specify the registers to save and restore.

In the implementation of the simple callee scheme for the experiment, a local variable is allocated to an available non-scratch register if the compiler estimates that the variable will be referenced at least three times. Similarly, an argument is allocated to an available non-scratch register if the compiler estimates that the argument will be referenced at least four times. An additional reference is required for arguments due to the initial load of the argument from the run-time stack to the register.

Simple Caller

The simple caller method places local variables and arguments in any allocable register. With this approach there is no notion of partitioning the available registers into scratch and non-scratch sets. The life of every available register does not extend across function calls and thus all registers are scratch. Any register that is used to hold a local variable or argument is saved immediately preceding a call and restored immediately following a call.

Unlike the simple callee method, the number of saves and restores in the simple caller method varies depending upon the estimated number of calls made by the function. In the implementation of the simple caller method, a local variable is allocated to an available register if *vpo* estimates that placing the variable in the register will result in fewer overall memory references. This occurs if the number of estimated references to the variable is greater than the estimated number of saves and restores (twice the estimated number of calls made). An argument is allocated to an available register if the number of references is estimated to be greater than the number of saves and restores plus one. As in the simple callee method, an additional reference is required for arguments due to the initial load of the argument from

the run-time stack to the register.

Simple Hybrid

How a register that holds the value of a variable can be saved and restored most efficiently depends on how the variable is used within a function. In some cases, it would be cheaper to allocate a variable to a callee-save register. This occurs when the life of the variable overlaps with call instructions. In other cases, it would be cheaper to allocate a variable to a caller-save register. This occurs when the life of the variable does not overlap with call instructions. Often both types of variables exist within a single function.

The simple hybrid method attempts to exploit these different lifetimes of variables by using a combination of simple callee and simple caller methods. The available registers are divided into two sets. One set, the non-scratch registers, is saved and restored by the simple callee method. The other set, the scratch registers, is saved and restored by the simple caller method.

First, *vpo* estimates the number of references that will be made to each variable. Then, for both methods, it calculates the cost of allocating the variable to a register (i.e. the cost of saving and restoring the register). Recall that for the simple caller method, the cost depends on the number of calls and their estimated frequency of execution. After all costs are determined, the variables are processed in order so that the variables with the highest reference count are processed first. A variable is allocated to a register from the set that minimizes the save/restore cost if such a register is available and performing the allocation reduces the number of memory references.

Smarter Callee

The smarter callee method is similar to the simple callee method with one difference. Instead of placing the saves and restores of non-scratch registers at the function entry and exit respectively, saves and restores are placed only around the region where the registers are used. For instance, if the lives of a non-scratch register are contained within a conditional, such as an if statement, then the save and restore of that register are also placed within the code generated for the if statement. If the execution path when the function is invoked does not enter this region of code, then the save and restore are not performed

unnecessarily. The implementation of this method, called shrink-wrapping, was accomplished by using Chow's data-flow equations[2] .

Smarter Caller

The smarter caller method is similar to the simple caller method except that data-flow analysis is used to minimize the number of saves and restores. This analysis allows two different optimizations to be performed. The first optimization eliminates saves and restores that are unnecessary. The second optimization attempts to move necessary saves and restores out of loops, thereby decreasing their frequency of execution.

The estimated number of saves and restores of registers allocated to an argument or local variable for the smarter caller method is determined in the following manner:

For each register at each call site in a function

(1) Determine if there is a potential *use* of the register after the call. If there is no *use* (only *sets* or returns as shown in Figure 1), then there is no need to save and restore the register associated with the call.

(2) Determine if there is a *use* of the register that follows the current call with no intervening call. If not (only *sets*, calls, and returns as shown in Figure 2), then there is no need to restore the register after the current call since it will be restored after a following call.

(3) Determine the cost of a restore before each *use* that follows the current call and is not preceded by another call. If these restores cost less than a restore following the current call, then place restores before each of these *uses*. Otherwise, place a restore immediately after the current call. These choices are illustrated in Figure 3. If there is a *use* of a register that can follow the current call and the *use* can also be reached without the current call being executed, then the restore is always placed after the current call. This is illustrated in Figure 4.

(4) Determine if there is a *set* of the register that precedes the current call with no intervening call. If not (only calls or function entry as shown in Figure 5), there is no need to save the register.

(5) Determine the cost of a save after each *set* that precedes the current call with no intervening call. Determine the cost of a restore after each call that precedes the current call with no intervening *set* of the register. If these saves following each *set* cost less than a save preceding the current call and the restores following preceding calls, then place saves after each of these *sets*. Otherwise, place a save before the current call and a restore following the preceding calls. This choice is shown in Figure 6.

The algorithm is first used to rank the local variables and arguments according to the estimated benefit of allocating the variable or argument to a register. Arguments and variables with the highest estimated benefit are allocated first. After variables have been allocated to registers and all optimizations have occurred, the algorithm is used to insert the appropriate saves and restores. Thus, the smarter caller method reduces the cost of saving and restoring registers in two ways. First, it saves and restores a

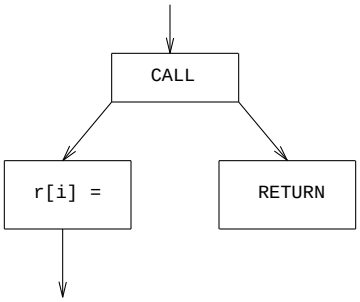

Figure 1. No Use of Register after Call

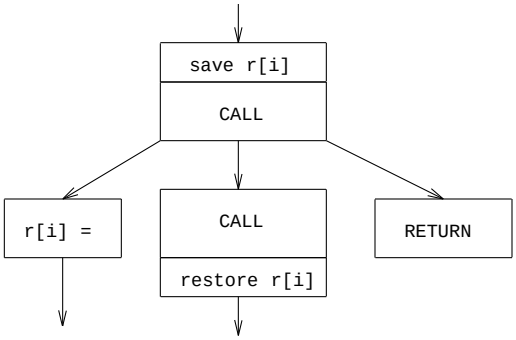

Figure 2. Only Sets, Calls, and Returns after Call

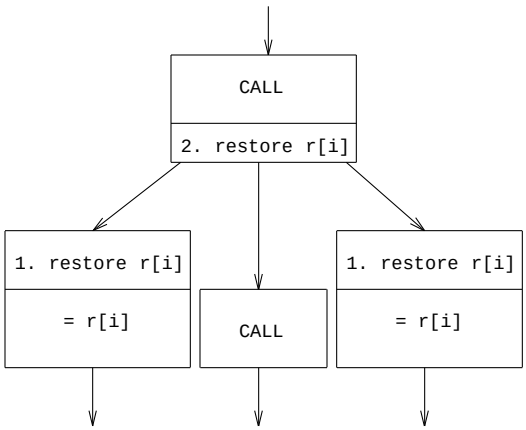

Figure 3. Restores before Uses (1) or after Call (2)

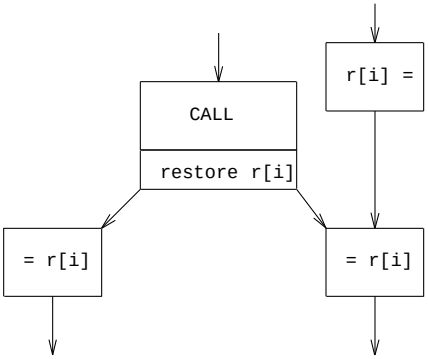

Figure 4. Use Preceded by a Set

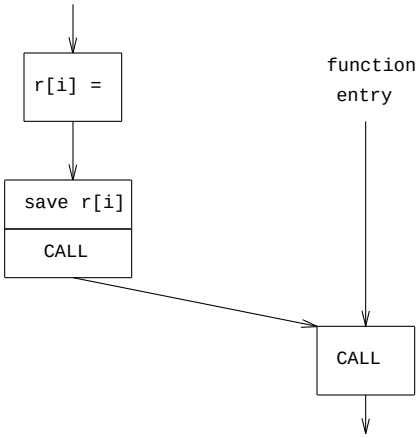

Figure 5. No Sets before Call

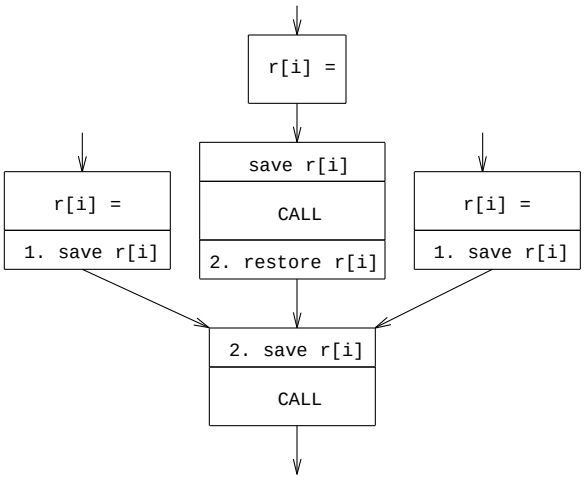

Figure 6. Saves after Sets (1) or before Call (2)

register only when necessary. Second, it moves the saves and restores to follow the previous *sets* and to precede the subsequent *uses* if it is beneficial. This has the effect of moving saves and restores out of loops.

Smarter Hybrid

The smarter hybrid method is similar to the simple hybrid method. The difference is that this method is a combination of the smarter callee and smarter caller methods. The available registers are partitioned into one set of non-scratch registers that is saved and restored by the smarter callee method and a second set of scratch registers that is saved and restored by the smarter caller method. *vpo* calculates the number of saves and restores as in both the smarter callee and the smarter caller methods. It then uses the method that has the greatest benefit for allocating a local variable or argument to a register.

PARTITIONING REGISTERS

Before one can evaluate the effectiveness of callee-save or hybrid calling sequences, the allocable registers must be divided into two sets. One set of registers, designated as non-scratch, are guaranteed to retain their values across calls, and the other set of registers, designated as scratch, are not. If the allocable registers in a callee-save calling sequence are partitioned such that there are too many scratch registers, then not enough variables can be allocated to registers. If there are too many non-scratch registers, then saves and restores of non-scratch registers used only as temporaries will be required. An inappropriate partitioning in a hybrid calling sequence can also result in poorer performance due to fewer variables being allocated to registers.

There has been little attention given to determining the best allocation of scratch and non-scratch registers in a callee-save calling sequence, and much variation exists. For instance, the fixed-point portions of the Digital Equipment Corporation's VAX-11 and the Harris Corporation's HCX-9 architectures are very similar. Yet the VAX-11 4.3 BSD C run-time library was implemented with six of its twelve user-allocable registers as scratch while the HCX-9 C run-time library was implemented with only two of its thirteen user-allocable registers as scratch. It appears that six registers were designated as scratch on

the VAX-11 since the `movc` instruction is hardwired to use `r0` through `r5`.

Both the callee and hybrid methods require that the available registers be partitioned into two sets. For these methods, the number of scratch registers was varied to determine the most effective combination of scratch and non-scratch registers. It was found on the VAX-11 that the most effective number of scratch registers from the twelve allocable general-purpose registers was five for the simple callee method, five for the smarter callee method, six for the simple hybrid method, and eight for the smarter hybrid method. The Motorola 68020 has eight allocable data registers, and six allocable address registers. The most effective number of scratch registers for this machine was discovered to be two data and two address for both the simple callee and smarter callee methods, three data and two address for the simple hybrid method, and four data and three address for the smarter hybrid method. The default number of scratch registers for each register type on this machine was two. Even though the C library on the SUN-3 could not be recompiled, the number of scratch registers can be increased.

To determine if the most effective percentages of allocable registers designated as scratch was independent of the number of available registers, *vpo* for the VAX-11 was modified to produce code assuming that the machine had four, eight, and sixteen user-allocable registers. Modifying the compiler to produce code with four or eight registers simply required specifying the reduced number of user-allocable registers to the register allocator. Using more registers than actually exist required only slightly more work. The nonexistent registers were associated with a set of local variables in each function. If a nonexistent register was referenced directly as an operand, then the corresponding local variable was used. If an instruction referenced a nonexistent register that was part of a more complex addressing mode, then the corresponding local variable was loaded into an available register preceding the instruction and the available register was referenced in the instruction instead. If there was a side effect from referencing the register, for example the autoincrement and autodecrement addressing modes, then additional instructions were generated to update the corresponding local variable. Six existing registers, the maximum number of registers that could be referenced in a single instruction for the VAX-11 *vpo* compiler, were also associated with a set of local variables. This ensured that there was always an available register for loading a variable that was associated with a nonexistent register.

Table 3 shows the results of varying both the number of user-allocable registers and the number of scratch registers for the simple callee, simple hybrid, and smarter hybrid methods. The smarter callee results are not shown since the simple callee results were approximately the same. Note that at least two scratch registers are required on the VAX-11 to return a double-precision floating-point value. These results imply that the number of user-allocable registers has little effect on the most effective percentage of scratch registers (about 40% for simple callee, 50% for simple hybrid, and 75% for the smarter hybrid). Only for the smarter hybrid with four registers did the percentage vary significantly.

Table 3: Results of Scratch/Non-Scratch Combinations

method	simple callee		simple hybrid		smarter hybrid	
user-allocable registers	scratch registers	memory references	scratch registers	memory references	scratch registers	memory references
4	2	98458465	2	96064606	3	93403276
	3	111117829	3	96354469	4	91751729
8	2	78974787	3	76202403	5	73310177
	3	78846670	4	75810477	6	72841135
	4	79774990	5	76581581	7	73308924
12	4	75512463	5	71894964	7	68958314
	5	75098651	6	71754355	8	68880184
	6	75752791	7	71968620	9	68987123
16	5	72855408	6	69337708	11	65653992
	6	72841124	7	69229771	12	65139854
	7	72924722	8	69302664	13	65593297

The desirable number of scratch registers seems to be greater than the number designated as scratch for many implementations of callee-save calling sequences. In several ports of *pcc*[10] , only two scratch registers are used even when additional scratch registers are available. There are instances on some machines where more than two scratch registers are required for calculations to avoid spills. Special instructions that require a number of registers, such as the move character instruction on the VAX-11, impact the available number of scratch registers. When no special instructions or calls that can update scratch registers are detected within a function, local variables and arguments can be allocated to scratch registers with no save/restore cost. The fact that over half the functions entered in the test set met

this criteria, also indicates the need for more scratch registers. By measuring the effect of varying the number of scratch registers, the appropriate number may be determined.

COMPARISONS OF THE DIFFERENT METHODS

Figures 7 through 12 show the results of using the six different methods on the two machines. For the callee and hybrid methods, which require the registers to be partitioned into scratch and non-scratch registers, the previously determined best combination was used. The number above each bar is the ratio for that type of measurement relative to the measurement obtained with the simple callee approach. When there was not room to place the ratio above the bar, it was placed below the bar.

The type of instruction or hardware support provided by the architecture to save and restore registers can obviously affect the results for a particular method. For instance, it would be an unfair comparison if the mask associated with the VAX-11 `calls` instruction was used for the simple callee method and a comparable strategy for the simple caller method was not employed. To determine the effect of the hardware support available on the performance of the six approaches, measurements were collected assuming three different hardware support mechanisms.

Measurements were collected assuming that 1) saves and restores are accomplished via the call instruction and a mask that indicates the registers to save and restore, 2) special instructions are available that can save and restore a set of registers on the run-time stack, and 3) saves and restores are done using primitive load and store instructions.

The first mechanism is often used on CISC machines. For example, the VAX-11 uses this mechanism to implement a callee-save calling convention. The second mechanism is also often used in CISC machines. The Motorola 68020's instruction set includes special instructions for saving and restoring the registers. For the measurements reported here, it was assumed that for each machine these instructions were four bytes in length. The actual length would depend on the implementation. The third mechanism, simple load and store instructions, would be used for a RISC machine (e.g., Motorola 88000 and MIPS R2000). Each load or store instruction required four bytes on both the VAX-11 and Motorola 68020.

In Figures 7 through 12, the solid lines represent the measurements obtained assuming that saves and restores are accomplished using the first mechanism. For each callee-save method, a mask is associated with each routine to save and restore non-scratch registers used in a function. With the caller-save conventions, a mask would be associated with each call instruction to save and restore the registers that are assigned to local variables and arguments. Since saves and restores in the smarter methods may be desired at locations other than the function entry/exit and call sites, this particular hardware approach is not appropriate for these methods. For the simple hybrid method, the caller mask is or'ed with the callee mask as a routine is invoked and entered. The registers corresponding to the resulting mask are saved along with the mask. When a return instruction is executed, the saved mask is loaded and the indicated registers are restored.

The dashed lines in Figures 7 through 12 represent the measurements obtained assuming the second mechanism, while the results for the third mechanism are shown using dotted lines. For the three smarter methods, it was assumed that these mechanisms would always save and restore a specific register in the same location for an invocation of a function since the set of registers being restored at a given point may not match the last set of registers saved. Since the number of memory references is the same for the second and third mechanisms, dotted lines are not shown in the figures showing the number of memory references (Figures 8 and 11).

Figures 7 through 9 display, for the VAX-11 the number of instructions executed, the number of memory references performed, and the static code size (sum of static code sizes for all programs in the benchmark suite). Table 4 in Appendix I shows the ratio of total memory references for the simple callee method to the five other methods for each program in the benchmark suite. Though the use of a mask associated with a routine or a call instruction results in the fewest instructions being executed for the three simple methods, it causes more memory references to be performed since for each function called two additional memory references are required to save and restore the mask.

For each measure, the simple callee and smarter callee methods produced similar results. There was only a 0.3% reduction in memory references using the smarter callee method. The smarter caller method, however, performed better than its corresponding simple implementation. In terms of

instructions executed, the simple caller, simple hybrid, smarter caller, and smarter hybrid are roughly equivalent. However, the smarter caller and hybrid approaches are clearly superior in reducing the number of memory references. It is interesting to note that there was a 4 to 3 ratio of restores to saves in the smarter caller method. More restores may occur in a loop when a call has *uses* and no *sets* of a register. While the restore has to remain in the loop, the save is placed before the head of the loop. Both the smarter caller and hybrid approaches reduced the code size by approximately six percent.

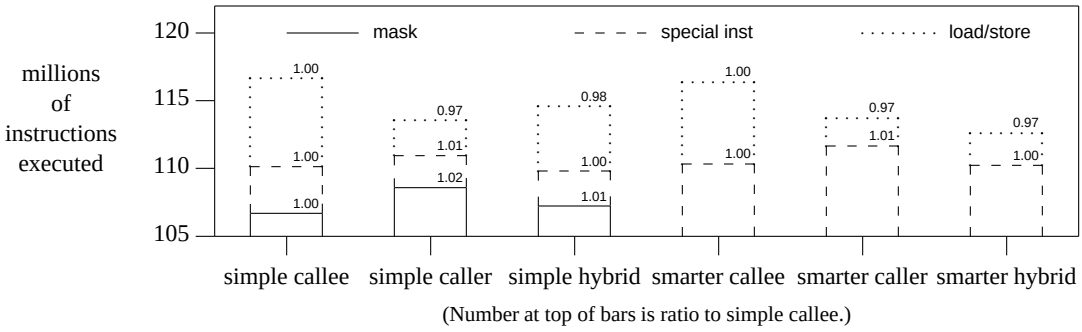

Figure 7. Total Instructions Executed - VAX-11

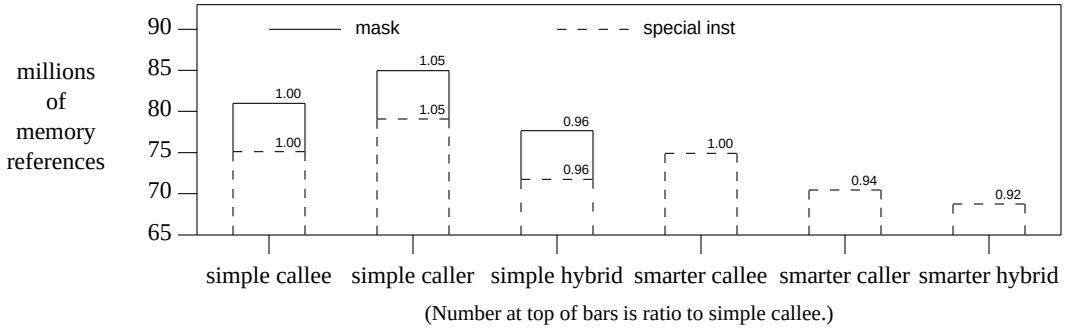

Figure 8. Total Memory References Performed - VAX-11

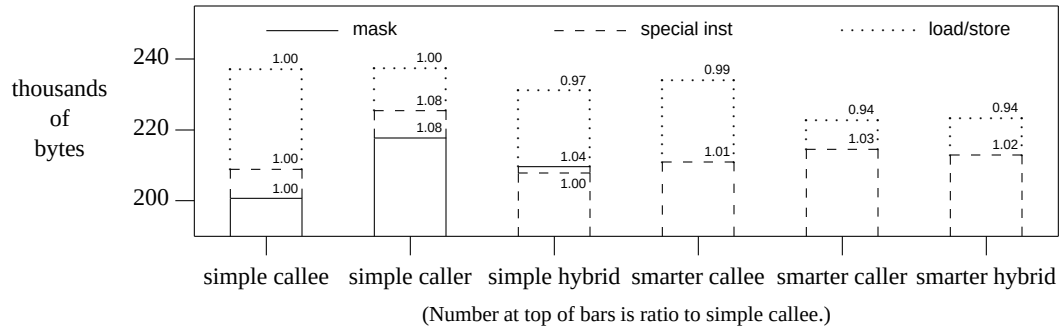

Figure 9. Static Code Size - VAX-11

Figures 10 through 12 display the number of instructions, number of memory references, and static code size for the Motorola 68020. Table 5 in Appendix II shows the ratio of total memory references for the simple callee method to the five other methods for each program in the measurement suite. The simple caller method resulted in the most instructions executed and the most memory references. Again, the improvement in the smarter callee method over the simple callee method was slight. Unlike the VAX-11, the simple hybrid was somewhat better than the smarter caller. The poorer performance of the caller methods for the 68020 occurred since the C library on the SUN-3, which contains many leaf functions, could not be used in data collection. The smarter hybrid method had the fewest instructions and memory references and required the least space.

The experimentation with various methods for saving and restoring registers gave some insight into which hardware mechanisms are most appropriate for saving and restoring registers. First, the measurements reported here indicate that the approach taken in the VAX-11 (a mask at the entry of the called function) results in poorer performance due to the requirement that the mask also must be saved and restored. Furthermore, this mechanism does not support shrink-wrapping.

Special instructions that save and restore a set of registers are more flexible and yield better performance. These instructions can be placed at the most appropriate site and the operand(s) that specify the registers to save or restore is part of the instruction and therefore does not need to be saved and restored. These special instructions also reduced the number of instructions executed in the smarter methods. This shows that even though data-flow analysis was used to place saves and restores at more beneficial loca-

tions in the code, many saves and restores still tended to cluster together.

The question of whether these special instructions should employ a mask or specify a range of registers to preserve is more problematic. The simple approaches could be implemented using the latter mechanism. The advantages are that the instruction may execute faster (no need to scan the mask) and the instruction can be encoded in fewer bits than an instruction that uses a mask. However, the smarter approaches favor the use of bit masks since it would be rare that a contiguous range of registers would be saved before a call (smarter caller) or when shrink-wrapping has been applied (smarter callee). Given the advantages of the smarter approaches, special instructions that use a mask seem to offer the most flexibility.

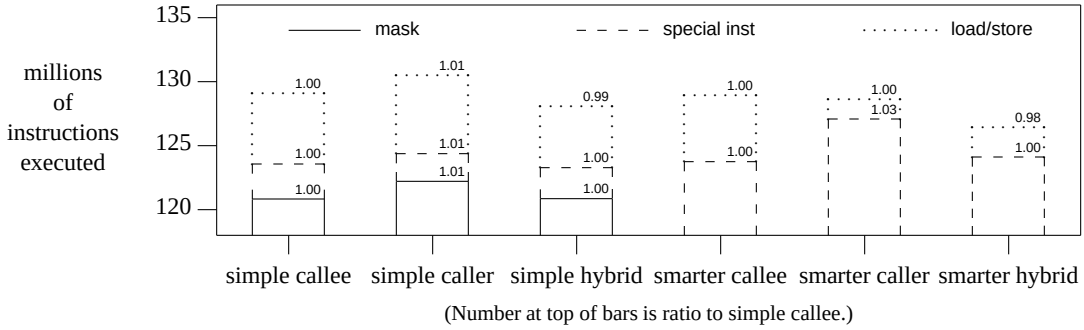

Figure 10. Total Instructions Executed - Motorola 68020

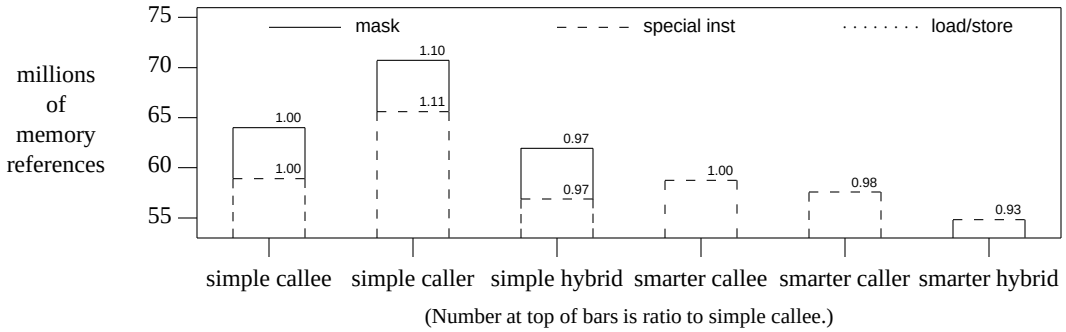

Figure 11. Total Memory References Performed - Motorola 68020

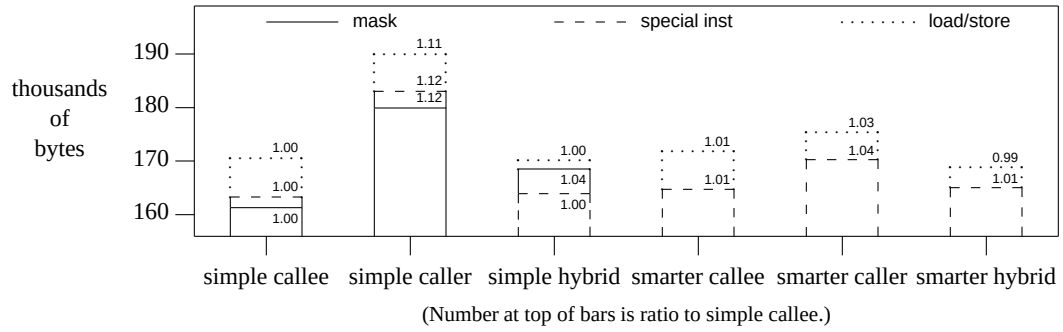

Figure 12. Static Code Size - Motorola 68020

While both of the machines used to evaluate the different approaches for saving and restoring registers across function calls are CISC architectures, some of the results should also be applicable to RISC architectures. The relative effectiveness of the six methods should be the same on RISC architectures. On the other hand, the results on the proper partitioning of the registers into callee- and caller-save sets is not applicable to RISC machines. RISC machines typically have 32 or more user allocable registers and tend to favor passing arguments through registers as opposed to on the run-time stack. These differences would certainly affect the allocation of registers to the two sets.

CONCLUSIONS

This study evaluated six different schemes for saving and restoring register values across function calls. The measurements show that the callee-save methods do not benefit from having data-flow information available. The measurements reported here are similar to those reported by Chow for shrink-wrapping applied to C programs[2] . Using data-flow information with the caller-save methods, however, was found to be very effective in reducing the number of memory references. For both machines studied there was over a 10 percent reduction in the number of memory references performed when the smarter caller method was used instead of the simple caller method.

The hybrid approaches produced better results than using a single method for saving and restoring registers. The results indicate that there are typically some situations where registers can best be saved and restored by a callee method, and other situations where the registers are best handled using a caller

save/restore method. The smarter hybrid approach produced the best overall results. Its implementation is only slightly more complicated than the smarter caller approach (if shrink-wrapping is not used) and our measurements showed that it resulted in the fewest number of instructions executed and the fewest number of memory references performed. It also produced the smallest code on the Motorola 68020. If speed of execution is the most important factor, then the smarter hybrid approach would be the method of choice. The simple hybrid method is an attractive choice if simplicity and compiler speed are the most important factors. While only slightly more complicated than the simple callee and simple caller methods, the simple hybrid approach produces code that is almost as effective as a smarter caller-save approach and it is much simpler to implement. Our production compilers use the simple hybrid approach.

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. National Science Foundation under grant CCR-8611653. Manuel Benitez implemented the machine-independent portion of *vpo* used to gather the measurements. The referees provided many valuable suggestions that measurably improved the quality of the paper.

REFERENCES

1. D. W. Wall, 'Global Register Allocation at Link Time', *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, 264-275 (June 1986).
2. F. Chow, 'Minimizing Register Usage Penalty at Procedure Calls', *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, Georgia, 85-94 (June 1988).
3. D. A. Patterson and C. H. Sequin, 'A VLSI RISC', *IEEE Computer*, **15**, 8-21 (September 1982).
4. M. Huguet and T. Lang, 'Reduced Register Saving/Restoring in Single-Window Register Files', *Computer Architecture News*, **14**, 17-26 (June 1986).
5. J. W. Davidson and D. B. Whalley, 'Ease: An Environment for Architecture Study and Experimentation', *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, Boulder, CO (May 1990).
6. D. B. Whalley, *Ease: An Environment for Architecture Study and Experimentation*, PhD Dissertation, University of Virginia, Charlottesville, VA, 1990.
7. M. E. Benitez and J. W. Davidson, 'A Portable Global Optimizer and Linker', *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, 329-338 (June 1988).
8. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins and P. W. Markstein, 'Register Allocation via Coloring', *Computer Languages*, **6**, 47-57 (1981).

9. F. Chow and J. Hennessy, 'Register Allocation by Priority-based Coloring', *Proceedings of the SIGPLAN Notices '84 Symposium on Compiler Construction*, Montreal, Canada, 222-232 (June 1984).

10. S. C. Johnson, 'A Tour Through the Portable C Compiler', *Unix Programmer's Manual, 7th Edition*, **2B**, Section 33 (January 1979).

APPENDIX I

Table 4: Relative VAX-11 Memory References to Simple Callee

program	simple caller	simple hybrid	smarter callee	smarter caller	smarter hybrid
cal	+1.0%	-1.3%	-0.2%	-3.1%	-5.2%
cb	-9.4%	-6.7%	-5.8%	-9.9%	-10.2%
dhrystone	-4.2%	-11.3%	0%	-7.6%	-11.1%
grep	+35.9%	-0.2%	-0.1%	-1.7%	-0.8%
matmult	+6.6%	-0.6%	0%	+6.5%	+2.9%
mincost	+20.1%	+0.7%	0%	+1.2%	-2.6%
od	+5.6%	-1.4%	0%	-12.8%	-14.4%
puzzle	-5.7%	-15.0%	0%	-12.1%	-15.7%
sieve	0%	-0.1%	0%	-0.4%	-0.4%
sort	+21.0%	-6.7%	0%	+7.4%	-8.5%
spline	+18.6%	-0.1%	0%	-7.4%	-3.6%
tr	-0.1%	-0.4%	-0.2%	-1.8%	-1.6%
wc	+7.0%	-0.6%	-0.2%	-3.1%	-2.7%
whetstone	-11.4%	-3.6%	0%	-21.1%	-17.0%
compact	+5.7%	-1.7%	0%	-0.1%	-2.7%
diff	+14.5%	-4.4%	-0.4%	+0.7%	-5.8%
nroff	+13.2%	-4.1%	-0.7%	-17.7%	-17.7%
sed	+7.2%	0%	-1.7%	-10.7%	-7.3%
vpcc	+11.7%	-3.0%	-4.1%	-11.8%	-12.3%
average	+7.2%	-3.2%	-0.7%	-5.6%	-7.2%
std dev	11.4%	4.1%	1.5%	7.5%	6.0%

APPENDIX II

Table 5: Relative 68020 Memory References to Simple Callee

program	simple caller	simple hybrid	smarter callee	smarter caller	smarter hybrid
cal	+1.9%	-0.8%	0%	-3.7%	-3.9%
cb	-17.3%	-3.4%	-5.8%	-17.4%	-15.4%
dhrystone	-0.9%	0%	0%	-8.4%	-15.4%
grep	+44.3%	0%	0%	-0.8%	+0.5%
matmult	+7.2%	0%	0%	+7.2%	+2.4%
mincost	+25.7%	-1.2%	0%	+9.0%	-1.2%
od	-2.4%	-9.8%	-2.9%	-16.4%	-10.8%
puzzle	-4.3%	-3.1%	0%	-11.0%	-6.5%
sieve	0%	0%	0%	0%	0%
sort	+27.4%	-12.5%	0%	+9.9%	-9.4%
spline	+19.5%	+1.7%	+0.1%	-9.3%	-5.1%
tr	+12.8%	-0.1%	-0.1%	-0.2%	-0.1%
wc	+44.1%	0%	0%	0%	0%
whetstone	-5.4%	0%	0%	-5.4%	-6.8%
compact	+11.7%	-0.8%	0%	-0.4%	-3.5%
diff	+23.7%	-1.1%	-0.3%	+8.6%	+0.3%
nroff	+13.6%	-3.5%	-0.5%	-17.7%	-19.9%
sed	+7.2%	0%	-0.8%	-9.8%	-4.9%
vpcc	+18.0%	-0.5%	-6.2%	-9.1%	-12.9%
average	+11.9%	-1.8%	-0.9%	-3.9%	-5.9%
std dev	15.9%	3.5%	1.9%	8.6%	6.3%