Fast Function Calls and Returns

Jack W. Davidson and David B. Whalley

Computer Science Report No. TR-90-20
August 1, 1990

Fast Function Calls and Returns

SUMMARY

A significant portion of the execution of a program is spent maintaining the state of each active function and in transferring information to and from functions. Many architectures include complex call, return, and argument passing instructions to save space, to reduce the total number of instructions, and to ease the job of the compiler writer. This paper describes the performance results obtained when simpler instructions are used to implement the calling sequence on a complex instruction set machine. First arguments, if possible, are passed through registers instead of the run-time stack. Next available complex call and return instructions are abandoned in favor of more primitive instructions. By using the more primitive call and return instructions in combination with passing arguments through registers, additional improvements on calls and returns are possible. These simple changes require no architectural modifications and are shown to have a significant improvement on the generated code.

KEY WORDS: Calling sequences Function call Function return Parameter Passing

INTRODUCTION

If a machine allows recursion, a run-time stack containing information about the state of each active function typically is maintained. The run-time stack can also be used to pass information between functions. Complex call, return, and argument passing instructions that make either implicit or explicit use of the run-time stack have been included in many architectures. Use of such instructions result in smaller programs and they reduce the number of instructions executed. Aside from any benefits that may result from simplifying the hardware implementation, the results presented here show that passing arguments through registers and using simpler instructions to perform function calls and returns provide significant performance improvements. In particular, the number of memory references performed is reduced significantly. Furthermore, such a calling sequence allows the compiler to perform a number of improvements that are not possible when complex instructions are used.

RELATED WORK

There has been much work in the past to attempt to speed up function calls and returns. A variety of instructions are available on different architectures to accomplish function linkage. Register windows have been used as a hardware approach to avoiding access to a run-time stack when the calling depth is shallow [Pat85]. Link-time optimizations have been used to avoid more complex call and return instructions in specific situations [BeD88]. Er [Er83] discussed three different schemes for optimizing function calls when the call is the logical last statement in the calling function. Powell [Pow84] stated that his

Modula-2 compiler produced code that executed faster when a more expensive procedure call mechanism was replaced with a simpler, faster one.

There has also been work in improving the performance of passing parameters to functions. In 1978 Tanenbaum discovered from dynamic measurements of 300 procedures in SAL that 72.1% of the calls executed had two or less arguments and 98% were passed in five or less arguments [Tan78]. A hardware approach to this problem is to allow parameters to be passed to routines in registers by use of overlapping sets of registers in register windows [Pat85]. Lisp compilers have been implemented with arguments passed in registers since the early 1980s [GrB82]. There have been several approaches to allow parameter passing through registers involving interprocedural analysis [BeD88, Cho88, Wal86].

ENVIRONMENT FOR EXPERIMENTATION

Most CISC machines have complex call, return, and argument passing instructions. To investigate the influence of modifications to a CISC calling sequence, an environment for experimentation was established on the VAX-11, a CISC architecture, to determine the effect of each modification. In this environment a compiler system, which performs several optimizations, was used. These optimizations include:

1. basic block reorganization
2. dead code removal
3. branch chaining
4. instruction selection
5. evaluation order determination
6. register allocation

This compiler was modified to record instruction information before it produced assembly code and to insert instructions to update frequency counters in the assembly code. These instructions were carefully placed to avoid their own contributions to the information they were meant to collect. For efficiency they were sparingly placed by recording the frequency of use of execution classes of basic blocks, rather than that of the basic blocks themselves. Two basic blocks are considered to be in the same execution class if each was executed every time the other was. Typically, the instrumented code ran 10 to 15 percent slower than code that was not instrumented.

The test set used for our environment consisted of eighteen programs distributed over forty-four files (more than 21,000 lines of C code). For each method, all source files of each program were recompiled on the VAX-11. 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 222 source files (including files from the C library). Data was collected from each of the files compiled by our compiler. The test set is summarized in Table I below.

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
	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

Table I: Test Set

PASSING ARGUMENTS THROUGH REGISTERS

The VAX-11 has twelve user-allocable registers (`r0-r11`). These registers are used to contain both integer and floating-point values. Double-precision floating-point values are contained in pairs of registers. These twelve registers are divided into two sets, scratch and non-scratch. Only the non-scratch registers are saved and restored at the function entry and exit respectively. Thus, scratch registers cannot be used across calls or special instructions, such as the `movc` instruction for moving blocks of characters, that can change the values of the registers.

Typically CISC machines, including the VAX-11, pass arguments on a run-time stack. The calling sequence for our VAX-11 compiler was modified to pass up to six parameters through the scratch registers. Double-precision floating-point parameters require two registers. At the point of the call, the compiler starts with the last parameter that would be pushed onto the run-time stack and stores it in a scratch register (`r5` if an integer, `r4` if a double). The compiler continues placing parameters in scratch registers until there are no more arguments or available scratch registers. Any parameters that cannot be passed through registers are pushed onto the run-time stack. The registers containing the arguments are stored in memory as a local variable at the function entry. Since the compiler knows the order of the arguments, it can store the correct register in the appropriate variable. These variables, like local variables, are candidates for being allocated to registers. This modification does not affect the availability of the scratch registers since they cannot be used across calls. This new scheme is illustrated in Figure 1.

```
        pushl   r8
        pushl   r9
        calls   $2,_foo

        =>

        movl    r8,r4
        movl    r9,r5
        calls   $0,_foo
        ...
.globl _foo
_foo:
        ...
        movl    r5,arg1.(fp)
        movl    r4,arg2.(fp)
```

Figure 1: Passing Arguments through Registers before Optimization

Using reference estimates, our compiler allocates the most frequently used variables to registers. Two memory references are saved if the parameter is allocated to a register. The memory references that are eliminated are the pushing of the value on the run-time stack in the calling function and loading the value from the stack into a register in the called function. If the parameter is not allocated to a register,

an additional instruction is required over the traditional scheme of using the run-time stack for passing parameters. The number of memory references in this case is unchanged. If the parameter is allocated to a register in a leaf function and that register is not used in the function, then the compiler will allow the parameter to remain in that register. In this case, one instruction is saved since there is no need to move the value of the register receiving the argument to another register.

A greater number of arguments should be allocated to registers using this scheme. The compiler will not allocate a local variable or parameter to a register unless the number of estimated references is greater than the cost of allocating the variable to a register. The save and restore cost of both local variables or parameters is two memory references. An additional cost of one memory reference is estimated for a parameter since the parameter has to be loaded from the stack into a register. Thus, the compiler requires that a parameter passed on the run-time stack to have at least four estimated references for it to be allocated to a register. Parameters passed through registers do not require loading from the run-time stack. An additional memory reference will also be found since the register containing the parameter is stored in memory at the function entry. Thus, a parameter passed through a register that is referenced only twice in a function can be allocated to a register by the compiler.

A number of special conditions had to be resolved to implement the new calling sequence. There are a set of routines which could not be compiled, since their source code was not available. These system calls were identified and parameters were always passed to these routines on the run-time stack. The set of routines in the C run-time library that accept a varying number of arguments on the run-time stack (e.g. printf, scanf, etc) were also identified. This set of routines depends on the calling sequence convention of how parameters are passed. For instance, they are dependent on the direction that the run-time stack grows and the order in which arguments are evaluated. Any parameter to these routines that was optional was passed on the run-time stack. Structure arguments are pushed on the run-time stack by the compiler using `movc` instructions. Thus, structure arguments in the new calling sequence were always passed on the stack.

USING PRIMITIVE CALL AND RETURN INSTRUCTIONS

Many machines have complex call and return instructions that perform a number of tasks. For instance, the VAX-11 `calls` and `ret` instructions save and restore the register mask, the program counter, the frame pointer, the argument pointer, and a value indicating the number of longwords of arguments. This value is used by the `ret` instruction to adjust the stack pointer back to the value prior to the caller pushing arguments onto the stack. The register mask is read by the `calls` instruction from the first word in the function and is used by the two instructions to save and restore allocable non-scratch registers.

Our VAX-11 compiler was modified to use more primitive call and return instructions. The modified compiler uses the VAX-11 `jsb` and `rsb` instructions which only save and restore the program counter. The `pushr` and `popr` instructions are used to save and restore allocable non-scratch registers, the frame pointer, and the argument pointer. The frame pointer and argument pointer, if used, are adjusted by add instructions in the called function. The stack pointer, adjusted by subtract instructions in both schemes at the function entry to allocate space for locals and temporaries, is restored by an add instruction in the new scheme before each `rsb` instruction. These changes are illustrated by the example in Figure 2.

Fewer data memory references should occur using the more primitive instructions. A large percentage of total memory references can be attributed to handling function linkage. In contrast to the `calls` and `ret` instructions, `pushr` and `popr` only save and restore the frame pointer and argument pointer if they are used. The stack pointer also only requires adjustment after a `jsb` instruction if an argument was passed to the called function. The frame pointer need not be adjusted if all locals are allocated to registers. More than 55% of the routines executed in the test set were leaves. Many of these leaves have all of its local variables allocated to scratch registers and thus will not require use of the `pushr` and `popr` instructions for saves and restores.

As in passing arguments through registers, some special conditions were required to implement more primitive call and return instructions. The set of routines not available to be compiled by *vpo* were identified and calls to these routines were generated with the `calls` instruction. Calls requiring the

```
        calls   $2,_foo
        ...
.globl _foo
_foo:
.word 0x0FC0
        subl2   $4,sp
        ...
        ret

        =>

        jsb     _foo
        addl2   $8,sp
        ...
.globl _foo
_foo:
        subl2   $4,sp
        pushr   $0x3FC0
        addl3   $32,sp,ap
        addl3   $32,sp,fp
        ...
        popr    $0x3FC0
        addl2   $4,sp
        rsb
```

Figure 2: Primitive Instructions for Implementing Calls and Returns

`calls` instruction accounted for a very small percentage of the total calls executed.

COMBINING BOTH METHODS

By using primitive call and return instructions, such as the `jsb` and `rsb` instructions, in combination with passing arguments through registers, some additional improvements to the code can now be accomplished. With the six scratch registers being used to pass arguments, very few of the compiled routines will use the argument pointer. Thus, saves and restores of the argument pointer is infrequent. Because arguments are passed through registers, typically at the point of the call the stack pointer is only adjusted by the `jsb` instruction to place the return address on the stack. Other optimizations can be accomplished when the call is immediately followed by an unconditional jump, the sequence of instructions to return to the caller, or another call.

A call followed by an unconditional jump can be optimized to avoid executing the unconditional jump. The `jsb` instruction pushes the address of the next instruction on the stack and jumps to the beginning of the function. To avoid the unconditional jump, the destination of the unconditional jump following the call can be pushed on the stack and then an unconditional jump to the routine can be used. When the called routine executes its `rsb` instruction, control will be transferred to the destination of the original unconditional jump. This is shown by the example in Figure 3.

```
jsb     _foo
jbr     L1

=>

pushl   $L1
jmp     _foo
```

Figure 3: Optimization of a Call Followed by an Unconditional Jump

A call followed by the sequence of instructions to return to the caller can be optimized to avoid the execution of the `rsb` instruction. The `rsb` instruction pops the return address off the run-time stack and branches to that address. To avoid execution of the `rsb` instruction, the sequence of instructions to return to the caller preceding the `rsb` are placed before the `jsb`, the `rsb` is removed, and the `jsb` is replaced by an unconditional jump. This results in the stack pointer being adjusted to point to the return address currently on the stack. Thus, two memory references are also avoided since there is no push and pop of the return address. This is illustrated in Figure 4.

Before attempting this type of optimization for a language such as Pascal, one must ensure that the called routine does not reference a variable declared in the calling routine. If such a variable is contained in the activation record of the calling routine, then its space should not be deallocated from the run-time stack until after the called routine references the variable.

A call to the current function followed by a return is known as tail recursion. The call and the sequence of instructions used to implement the return can be replaced by a branch to the function entry

```
jsb     _foo
popr    $0x40C0
addl2   $40,sp
rsb

=>

popr    $0x40C0
addl2   $40,sp
jmp     _foo
```

Figure 4: Optimization of a Call Followed by a Return Sequence

following any instructions used to save the state of the caller. This is illustrated in Figure 5.

```
.globl _foo
_foo:
        subl2   $4,sp
        pushr   $0x20C0
        addl3   $12,sp,fp
        ...
        movl    r6,r5
        jsb     _foo
        popr    $0x20C0
        addl2   $4,sp
        rsb

        =>

.globl _foo
_foo:
        subl2   $4,sp
        pushr   $0x20C0
        addl3   $12,sp,fp
LB1:
        ...
        movl    r6,r5
        jmp     LB1
```

Figure 5: Optimization of Tail Recursion

Sometimes a sequence of calls occurs with no intervening instructions. Instead of returning to the calling routine after executing each called routine, control can be transferred directly to the beginning of the next routine to be invoked in the sequence. To avoid executing these calls, the address of the instruction following the last call is pushed on the stack, the address of each call except for the first is pushed on the stack in reverse order, and an unconditional jump is made to the first function called. When the first function executes its `rsb` instruction, control will be transferred to the beginning of the second function to be called since its address was on the stack. When the last function executes its `rsb` instruction, control will be transferred to the instruction following the last call. An example of this optimization is shown in Figure 6.

```
        jsb     _foo1
        jsb     _foo2
        jsb     _foo3

        =>

        pushl   $L1
        pushl   $_foo3
        pushl   $_foo2
        jmp     _foo1
L1:
```

Figure 6: Optimization of a Sequence of Calls

To enable the three types of optimizations with calls to be performed more frequently, instructions immediately following a call were attempted to be moved before the call. This will result in calls being followed by unconditional jumps, return sequences, and other calls more often. An instruction can only be moved to precede a call if it doesn't:

1. adjust the stack pointer
2. reference a scratch register
3. reference a global variable
4. reference a variable that has been used indirectly
5. set condition codes that will be used
6. change the program counter

RESULTS

Table II compares the results of running the test suite using four different versions of the compiler.

measurement	default	percentage change from default		
		parameters thru registers	primitive instructions	params thru regs and primitive insts
instructions	90,247,740	+1.5%	+11.6%	+6.0%
calls	2,833,438	-0%	-17.0%	-17.0%
transfers of control	27,308,488	-0%	-0%	-1.6%
func linkage memrefs	22,300,144	-0%	-50.8%	-68.0%
total memrefs	84,403,874	-8.4%	-13.5%	-26.0%

Table II: Results from the Four Calling Sequences

These results show that passing arguments through registers can effectively reduce the number of memory references. This simple change to the calling sequence convention resulted in 8.4% fewer memory references. The reduced number of memory references resulted from not having to push arguments onto the run-time stack and referencing those arguments from the run-time stack. This savings was possible due to an increase from an average of 0.58 to 2.21 arguments allocated to registers per function invoked. There was also 1.5% more instructions executed. Use of a link-time optimizer, such as VLINK [BeD88], could remove these extra instructions.

The results in Table II also show the benefits of using more primitive call and return instructions. By not performing all of the functions associated with the more complex call and return instructions, there were 13.5% fewer memory references. There was also 11.6% more instructions executed. These additional instructions are simpler and less costly than the fewer complex instructions they replaced.

More improvements were accomplished from the combination of passing arguments through registers and using primitive call and return instructions. Most of the 4.7% fewer memory references were due to not having to save and restore the argument pointer when no arguments were passed on the stack in calls to other functions. The 7.1% fewer executed instructions occurred since there were fewer save and restore instructions, fewer instructions to adjust the stack pointer after a call with arguments, and fewer adjustments of the argument pointer for a function that received arguments. Using the new optimi-

zations and movement of instructions before calls, 17.0% of calls were optimized into other instructions. Of these calls that were optimized, 45.8% were followed by a return sequence, 31.3% were followed by an unconditional jump, and 22.8% were followed by another call. Tail recursion optimizations, which have received some attention in the past, was found to occur very infrequently in the test set. Compilation techniques for other paradigms may apply tail recursion more often. For instance, some Scheme compilers translate the source code into an intermediate form, continuation-passing style, where tail recursion is explicit [KKR86]. The 0.6% reduction in memory references performed and the 0.2% reduction in instructions executed by applying these optimizations was not significant. The 1.6% reduction in the number of transfers of control has more of an effect since branches can result in instruction pipeline stalls.

CONCLUSIONS

Passing parameters in registers has been recognized as beneficial in the past. There has been a variety of schemes used to implement this feature including register windows and link-time optimizations. Registers windows, though effective, has many disadvantages that include the area required on the chip for the large number of registers, the increase in instruction cycle time due to longer access to a register, and increased process switching time [Hen84]. Link-time optimizations have been used to adjust code to pass an argument through a register instead of the run-time stack when the argument is allocated to a register [BeD88, Wal86]. This experiment has shown that almost four times as many arguments are allocated to registers when arguments are passed through registers as a calling sequence convention. By simply changing the calling sequence a significant improvement can be obtained without expensive hardware or software.

A few recent RISC machines pass arguments through registers as a calling sequence convention. Most of these machines allow only a subset of the available registers for passing arguments. For instance, the Clipper only allows at most two arguments to be passed through registers. This experiment has shown that all of the scratch registers in a callee-save calling sequence can be used to pass arguments without affecting the availability of these registers.

Maintaining the state of each active function on a run-time stack has been recognized as expensive in the past. To address this problem a variety of complex instructions have been implemented on different machines. Complex function call and return instructions have been recognized as both frequently occurring and time-consuming [Wie82]. It has been shown that many of the functions performed by these complex instructions are unnecessary a large percentage of the time. Saving and restoring the environment registers on the VAX-11 in this study accounted for approximately a fourth of the total memory references. By using the primitive instructions about one half of the memory references for saving and restoring environment registers or about one seventh of the total memory references were avoided.

By using the primitive instructions along with passing arguments through registers, more improvements to the generated code were accomplished. Since arguments were rarely passed on the run-time stack, adjustments of the pointer to the run-time stack after a call, adjustments of the argument pointer, and saves and restores of the argument pointer could be avoided. New optimizations were available on 12% of the calls. By moving instructions that follow a call to instead precede the call, when it is possible, it was found that these optimizations were available on 17% of calls.

By simply changing the calling sequence to pass arguments through registers and use more primitive instructions a significant improvement can be obtained. For instance, dhrystone, a call-intensive program, had an improvement of 7537 dhrystones (6 seconds) versus 5952 dhrystones (8 seconds) using the version of the compiler with no calling convention modifications.

REFERENCES

[BeD88] 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, June 1988, 329-338.

[Cho88] F. Chow, Minimizing Register Usage Penalty at Procedure Calls, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988, 85-94.

[Er83] M. Er, Optimizing Procedure Calls and Returns, *Software—Practice & Experience 13*,10 (October 1983), 921-939.

[GrB82] M. L. Griss and E. Benson, Current Status of a Portable Lisp Compiler, *Proceedings of the SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 276-

283.

[Hen84] J. L. Hennessy, VLSI Processor Architecture, *IEEE Transactions on Computers 33*,12 (December 1984), 1221-1246.

[KKR86] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams, ORBIT: An Optimizing Compiler for Scheme, *Sigplan Notices 21*,7 (July 1986), 207-218.

[Pat85] D. A. Patterson, Reduced Instruction Set Computers, *Communications of the ACM 28*,1 (January 1985), 8-21.

[Pow84] M. L. Powell, A Portable Optimizing Compiler for Modula-2, *Proceedings of the SIGPLAN Notices '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, 310-318.

[Tan78] A. S. Tanenbaum, Implications of Structured Programming for Machine Architecture, *Communications of the ACM 21*,3 (March 1978), 237-246.

[Wal86] D. W. Wall, Global Register Allocation at Link Time, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 264-275.

[Wie82] C. A. Wiecek, A Case Study of VAX-11 Instruction Set Usage for Compiler Execution, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March, 1982, 177-184.