Register Allocation for Free: The C Machine Stack Cache

David R. Ditzel

Computing Science Research Center
Bell Laboratories
Murray Hill, New Jersey 07974

H.R. McLellan†

Massachusetts Institute of Technology
Cambridge, MA 02139

Introduction

The Bell Labs C Machine project is investigating computer architectures to support the C programming language.[1] One of the goals is to match an efficient architecture to the language and the compiler technology available. Measurements of different C programs show that roughly one out of every twenty instructions executed is either a procedure call or return.[2] Procedure call overhead is therefore a very important consideration in the overall machine design. A second and related area of primary concern in overall machine efficiency is the register allocation strategy. While use of additional registers can offer considerable improvement in execution times, adding registers usually has the adverse effects of increasing the procedure call overhead due to register saving and creating an undue burden on the compiler. In this paper we describe a piece of the C Machine architecture which effectively eliminates the register allocation problem, and improves procedure calling by drastically reducing storage references required by traditional register saving. The technique can be generalized for other languages and architectures, though we will only directly address those issues involving the C language.

Registers: Why

Small numbers of general purpose registers have been used for many years for the following reasons. The small size and physical proximity of registers to the ALU offers access times often an order of magnitude less than main memory. Direct addressing with a small number of bits is faster than, and uses less hardware than cache memories. The small size of the register index field leads to compact code. Registers provide a physically separate memory that can be accessed in parallel with main memory. Because they are small in number, registers can be cheaply duplicated to allow the simultaneous fetching of two operands. By reducing accesses to main memory, registers decrease the memory bandwidth requirements of the CPU.

Registers: Why Not

The use of registers also has several drawbacks. Registers must usually be saved across procedure calls; this saving tends to make procedure calls time consuming. Expensive procedure calls inhibit structured programming techniques that encourage many small procedures. To achieve any benefit, registers must be efficiently allocated. The predominant use of high level languages places the burden of register allocation on compilers. Effective register allocation usually requires multiple passes by large complex compilers. Multiple pass compilers are slow, expensive and hard to debug. One would prefer that a machine architecture encourage simpler compilers rather than more complicated compilers. Registers are not equivalent to memory; one can generally not take the address of a register, nor do addressing modes apply to registers. One can not place an array or string in registers and use them in a manner consistent with the semantics of primary memory. Registers are usually limited to holding data the size of the machine word. Compilers for the C language tend to be one pass and do not allocate variables to registers beyond the expression level. The only exception to this simple allocation is the explicit hint supplied by the programmer in using the C keyword "register". The reason for this is that a one pass compiler otherwise has no a priori knowledge that a variable is not aliased,[3] i.e. referenced through a pointer. A dereferenced pointer would yield the old value in memory rather than the new value in the register, hence variables may not be left in registers.

Despite these problems, the use of high speed registers is still attractive and efforts have been made to reduce the

† Work performed while at Bell Laboratories.

drawbacks. In fact, recent developments in VLSI and high speed memories are driving architectures toward large numbers of registers. Sites[4] advocates using either a renaming mechanism or banks of registers for efficiently using 100 to 1000 registers. Dannenberg[5] proposes using many registers for holding local variables in block structured languages, but does not address the problem of aliasing or compiler complexity. The BBN C/70 [6,7] has 1024 registers that it divides into register sets, allocating a new set of 8 registers for each procedure call. The register sets are treated as a circular buffer and flushed incrementally to memory if the buffer overflows. The RISC I microprocessor[8] uses a similar circular buffered register scheme, but can overlap its internal register banks in a manner similar to that of the BELLMAC-8[9,10] microprocessor's external registers.

Why Not Replace Registers with a Cache?

We had at one time considered replacing the traditional general registers with a cache, but duplicated as one would duplicate register sets to allow parallel multiple operand access. Caches are fast memory, and in fact one of their strong points is that **they are memory,** and as such do not have fixed width or addressability problems. But caches are never as fast as registers for the following reasons. To start with, cache accesses are usually the result of some addressing mode calculation, such as adding the Stack Pointer to an offset; registers are directly addressed without this additional calculation. Before being presented as valid, the cache address tag must be compared to the submitted address. This comparison process cannot begin until the data would be considered valid for a register access. Having more than one entry per cache line is also time consuming as the data must pass through a multiplexor as well. Using only one entry per cache line eliminates the multiplexing, but is expensive in terms of nearly doubling the amount of hardware required for an equivalent size register set as there need to be as many tags as data entries. Additional control is also needed with caches for updating the replacement information and handling misses, adding hardware in a time critical area of the machine. For an ECL implementation of the C machine, the tag comparison plus multiplexing would be roughly comparable to the memory access time, leading to the conclusion that registers would yield nearly twice the performance of a real cache.

Registers have additional benefits over caches, even if the access times could be made equivalent. Referencing newly allocated variables in a cache causes the cache to miss for the first access, whereas allocation of a new register causes no miss. This is particularly important for something as dynamic as the allocation of local variables on a stack, and is significant enough that the IBM 801 computer[11] has separate instructions to allocate and free cache lines. Also, because cache memories perform a many-to-one address mapping, conflicts can occur that cause the cache to have a very low hit rate. Registers always hit.

A Fast C Calling Sequence

The procedure calling sequence has a large impact on the performance of any machine; for this reason it is important to make the calling sequence efficient. A calling sequence includes copying the outgoing arguments to an argument area, saving the return address, transferring control to the called procedure, and allocating space for the new procedure's local variables, temporaries and outgoing arguments. The calling sequence for the C Machine accomplishes these goals with as few operations as possible using a scheme similar to that of an earlier C Machine study.[12,13] A typical procedure stack frame is depicted in Figure 1. The stack grows downward in memory, with the stack pointer (SP) always pointing to a free memory location. This free slot is where the program counter (PC) will be stored on a procedure call (or interrupt). This avoids having to adjust the stack pointer to save or restore the PC. The PC is the only machine register which is implicitly saved on a procedure call. Above the address of the SP in the stack frame is a space large enough to store all outgoing arguments for any call from the current procedure. Above the outgoing arguments is space for temporary values and local variables.

The highlight of this scheme is that the SP is modified only on procedure entry or procedure return. Traditional calling sequences[14] that store local variables above the SP and outgoing arguments below the SP need to modify the SP at least an additional time in allocating space for the arguments. Having *push* or *pop* operations that automatically adjust the SP was intentionally avoided. After a traditional *push* or *pop* instruction, operand address generation for subsequent instructions must wait for the SP to be modified. The relatively constant SP eases the flow of instructions through a highly pipelined CPU.

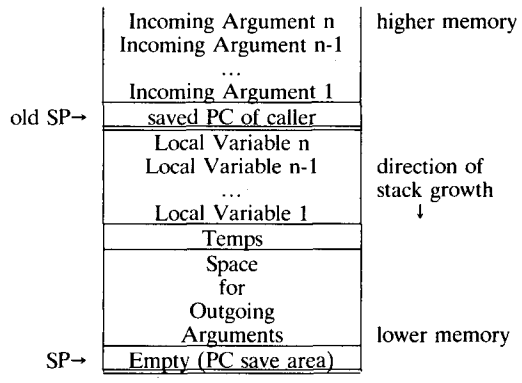


Figure 1. Stack Frame Layout

The steps required for a procedure call are straightforward. Arguments are *moved* onto the stack, with argument1 stored at SP+1, argument2 at SP+2, etc. Note that this is not a traditional push operation, as the SP is not modified as each argument is put on the stack. The **call** instruction saves the return point and then changes the PC to the address of the first instruction of the called procedure. The first instruction of the called procedure adjusts the SP to allocate its own stack frame. The last instruction of the called procedure, **return,** re-adjusts the SP to deallocate its stack frame and then branches to the address pointed to by the SP.

The C Machine Stack Cache

The goal of the Stack Cache is to keep the top elements of the stack in high speed registers. The problem we solve is how to perform the allocation of these registers without placing the burden on the compiler, and at the same time retaining the efficiency of register accesses. Control of the hardware, the instruction set, and a disciplined use of the aforementioned calling sequence allows a memory-to-memory style architecture (i.e. registerless to the compiler) to perform an automatic binding of memory addresses to machine registers. This dynamic binding is particularly advantageous when used with the instruction cache in the C Machine. The result is that accesses to most local variables, temporaries, and incoming and outgoing arguments will have the speed and simplicity of register references.

Before describing the dynamic binding, a description of the Stack Cache register set design is needed. The Stack Cache register set is a circular buffer maintained by two registers, the Maximum Stack Pointer (**MSP**) and the Stack Pointer (**SP**). Both the MSP and the SP are 30 bit registers holding word addresses. The MSP is used to delimit the highest address of data which is currently kept in the Stack Cache registers; the SP delimits the lowest address of data in the Stack Cache. The Stack Cache registers themselves are ordinary high speed random access memory devices. Individual Stack Cache entries are 32 bits in size (the wordsize of the machine) and the number of entries must be a power of two. Although the Stack Cache register block is word addressed, halfwords and bytes can be read as well by using the two lowest bits of a byte address to control a multiplexor to sign extend and align the output. Data is always aligned so that word and halfword references do not cross word boundaries. Byte addressability is important in eliminating the dichotomy of registers appearing different from memory to the compiler, as the machine itself is byte addressable. Figure 2 shows the rudiments of a simple Stack Cache with 1024 32-bit registers.

Two key principles are involved in allocating the registers of the Stack Cache. First, as instructions are prefetched, all operands addressed with the form **SP+offset** are computed to form a 32-bit address. This partially decoded instruction is stored in the Instruction Cache, rather than storing instructions with offsets. Second, since our calling sequence guarantees that the SP will not change during the activation of a particular procedure, it can also be determined at prefetch time whether the data at a particular address will be resident in the Stack Cache registers, or if it will be in main memory. If the address lies between the SP and the MSP, then the data will be resident in the Stack Cache registers and the addressing mode of the operand is changed to register before the instruction is stored in the Instruction Cache. Because the size of the Stack Cache is a power of two, the register number is simply the low *log2(SizeOfStackCache)* bits of the word address. In essence, all local variables have been allocated to registers.

One advantage of placing a virtual address in the instruction cache rather than an offset is that at least one pipeline stage can be eliminated. Every time such an instruction is referenced from the instruction cache a stack pointer plus offset calculation has been saved, in exactly the same manner as conventional registers. Unlike traditional general register sets, the Stack Cache has the properties of main memory, as would a traditional cache. In particular, the registers are byte addressable, reflect the semantics of memory, and as far as the compiler is concerned, are not limited in number. Unlike traditional caches, the Stack Cache registers hold contiguous words of memory, and are far less costly to implement in terms of both complexity and circuit density. As measurements later show, the technique is remarkably effective for real programs. Reentrancy for cached instructions is guaranteed by including the SP as part of the address tag in the Instruction Cache.

Integration with the Instruction Set

Four instructions cooperate in maintaining the Stack Cache. They are **call, return, enter** and **catch.** The **call** instruction takes the return address and saves it on the stack, then branches to the target address. The target of a **call** instruction must be an **enter** instruction. The **enter** instruction is used to allocate space for the new procedure's stack frame by subtracting its operand, the size of the new stack frame in machine words, from the SP. The **return** instruction deallocates the space for the current stack frame by adding its operand to the SP, then branching to the return address on the stack. The **catch** instruction is always the target of a return instruction, and is used to guarantee that the Stack Cache is filled at least as deep as the number of entries specified by the operand of the **catch** instruction.

Enter and **catch** are also used to handle the cases where the Stack Cache register set is not large enough to accommodate the entire stack. When a new procedure is entered, the **enter** attempts to allocate a new set of registers equal to the size of the new stack frame. If free register space exists in the circular buffer for the new stack frame then all that needs be done is to modify the SP. If not, then the entries nearest the MSP are flushed back to main memory. Two cases exist:

- If the new frame size is less than the size of the Stack Cache then only the frame size minus the number of free entries must be flushed.
- If the new frame size is greater than the size of the entire Stack Cache, then all active entries are flushed and only part of the new frame nearest the SP is kept in the Stack Cache.

Upon procedure return it is not known how many of the Stack Cache entries had to be flushed since the call, so some entries may need to be restored from primary memory. The argument of the **catch** instruction specifies the number of Stack Cache entries that must be valid before the flow of execution can resume. **Catch** is the guarantee that Stack Cache references can be bound to register numbers and accessed without having to check to see if the data is actually resident in the Stack Cache. Unlike normal caches, and like registers, a Stack Cache reference will never miss. The **catch** instruction performs much like an assertion, in that most of the time no entries will need to be restored from memory.

Stack Depth Analysis

The use of the Stack Cache is most effective when the stack tends not to grow or shrink rapidly or by large amounts. To measure stack behavior a VAX C compiler was modified to produce extra code around each procedure call and procedure entry to record the stack depth.[15] The data and abbreviated plots for four well known programs are presented here. First is *pcc2*, a version of the portable C compiler, compiling the machine dependent part of the compiler. Second is the *troff* text formatter processing the troff tutorial. Third is the code generation pass of the PDP-11 compiler, compiling the file "c10.c". The PDP-11 compiler was chosen because its recursive descent approach to code generation was expected to use the stack heavily. Fourth is a VLSI design rule checker, *dr*, checking a 32-bit adder. Rather than presenting volumes of numerical data, the stack depth was plotted on a dot matrix printer. Even with 100 dots/inch resolution, printing half a million calls using one dot per call would take nearly 600 pages. For this reason a number of sequential calls are "folded" to fit on the same x coordinate. The depth of the stack in 32-bit words is shown by the y coordinate. Table 1 summarizes the results numerically. The amazing result is that with only 1024 registers, three of the four programs would never exceed the size of the Stack Cache at all! By looking at the plots it is evident that even if the Stack Cache size was considerably smaller than 1024, the overhead involved in flushing and restoring parts of the Stack Cache would be very small. The plots show that the stack depth typically does not vary rapidly, nor does it vary by large amounts. To further substantiate that this behavior is typical, the sizes of new frames allocated at procedure entry were measured for all the standard UNIX† commands (/usr/src/cmd/*.c) with a compiler generating code for the VAX. The frame sizes are measured in 32-bit words and include all outgoing arguments, local variables, temporaries and the PC save area, using the previously described calling sequence. Table 2 summarizes the frame sizes for these 188 commands containing 1449 procedures in 53,197 lines of C code. Based upon these measurements we conclude that typical program behavior is ideal for exploitation by the Stack Cache.

Table 1. Summary of Stack Depth Measurement

Plot Number	Program	Number of calls	Maximum Depth	#Calls/dot on plot
1	pcc2	633,540	864	1000
2	pdp-11 cc	124,351	1106	200
3	troff	104,098	91	200
4	dr	548,023	459	1000

Table 2. Distribution of Frame Sizes for Unix Commands.

Size	Occurrences	%	Cumulative%
1	58	4.0	4.0
2	134	9.2	13.3
3	194	13.4	26.6
4	220	15.2	41.8
5	201	13.9	55.7
6	110	7.6	63.3
7	113	7.8	71.1
8	57	3.9	75.0
9	48	3.3	78.3
10	40	2.8	81.1
11	31	2.1	83.2
12	14	1.0	84.2
13	8	0.6	84.7
14	13	0.9	85.6
15	15	1.0	86.7
16-31	64	4.4	91.1
32-63	33	2.3	93.4
64-127	17	1.2	94.5
128-255	57	3.9	98.5
256-512	12	0.8	99.3
>512	10	0.7	100.0

† UNIX is a Trademark of Bell Laboratories.

Memory Traffic Analysis

One of the most important uses of registers is in reducing memory traffic, thereby increasing the performance of a machine. The dynamic frequency of addressing modes can be used to examine the effect of registers on memory traffic, with a little knowledge about the compiler's code generation strategy. On the VAX, for example, all arguments are referenced with the Argument Pointer register, all locals from the Frame Pointer, etc. Table 3 shows the breakdown of data memory references for the standard C compiler for the VAX compiling the C Machine compiler.

In Table 3, the category "registers" does not include references to the Stack Pointer, Argument Pointer, Frame Pointer or Program Counter. "Locals" includes references to local variables on the stack. "Globals" includes references to all statically allocated data. "Argument refs" are the references within a procedure to its incoming arguments. "Argument pushes" is the traffic generated by the calling procedure in storing outgoing arguments on the stack; fetching the arguments is included in the other types. "Indirect" references count all memory accesses where data is fetched through a pointer. For indirect references it is not possible to precisely determine the type of the data being fetched, however, experience suggests that most of these references are likely to be to the global area, particularly structures or arrays. Counts are provided as a percentage of all operand references and as a percentage of data memory references. Note that each operand may make more than one reference to memory (for example, indirecting through a local variable would count for both a reference to a local and an indirect). References to immediate data, which were measured to be 21% of addressing mode usage on the VAX, are not included since they cause no separate storage access.

By being clever with a compiler's code generator, we also measured how the C Machine's 2-½ address memory-to-memory style architecture is likely to perform. A C compiler was constructed that mimicked the C Machine compiler by producing VAX code which did not utilize the general registers and copied the aforementioned calling sequence. In most cases either one or two VAX instructions were sufficient to mimic a single C Machine instruction. This accounts for a slight increase in the number of instructions. Table 4 shows the results for running this "Registerless VAX" compiler on the VAX, compiling the same program as for Table 3.

Since all locals and argument references are likely to turn into Stack Cache register accesses, as shown by the stack depth analysis, Table 4 indicates that 82% of all data references would be to registers. Table 5 relates these numbers to the VAX to show how significant the savings can be. Data storage accesses are reduced by a factor of 3.3 over the standard VAX C compiler and by a factor of 5.7 over a true memory-to-memory style architecture. The counts presented in Table 3 do not include the rather substantial traffic incurred on the VAX in saving and restoring registers for each call and return, so the overall data memory traffic savings are likely to be more optimistic than the tables indicate.

Table 3. Operand Usage for VAX

Type	Count	%Total	%Memory
Registers	165,646,997	35%	-
Locals	71,744,275	16%	24%
Argument refs	96,364,353	21%	32%
Argument pushes	34,443,896	7%	11%
Globals	14,384,977	3%	5%
Indirect	84,788,363	18%	28%
Total	467,372,861	100%	100%
Instructions Executed	391,830,910		

Table 4. Operand Usage for Registerless Machine (VAX)

Type	Count	%Memory
Locals	318,984,183	57%
Argument refs	109,328,303	19%
Argument pushes	34,474,782	6%
Globals	14,518,861	3%
Indirect	85,117,089	15%
Total	562,423,232	100%
Instructions Executed	420,725,252	

Table 5. Average Memory Traffic Per Instruction

0.77	Memory references per instruction with standard vax compiler
1.34	Memory references per instruction on "registerless" vax
0.24	Memory references per instruction with Stack Cache registers

Global Registers

We have used the words "Stack Cache" because the interesting part of the mechanism is in the early binding of stack offsets and in assigning register numbers automatically. But there is no restriction from using a similar mechanism to cache any particular piece of memory. Substantial benefits can be gained from allocating a small number of registers for global variables. For example, in addition to the Stack Cache registers one could include 16 words of memory that were mapped into memory locations 0 through 15. Even large programs tend to use relatively few global variables, and a small percentage of these account for most of the uses. For example, *pcc, troff* and *dr* have 95, 234 and 79 scalar global variables, respectively, of which 90% of the dynamic references occur in only 16, 46 and 18 global variables. Of all references to absolute addresses, scalar global variables account for 91%, 96% and 94%, respectively.

Supporting Compilers

Making the internal machine registers invisible to the compiler has several benefits. Code generation is considerably eased because no register allocation is required. In particular, a large optimizing compiler is not needed to perform efficient register allocation; a simple one pass compiler can have comparable performance. Because the compiler will be smaller and easier to write, it is more likely to be free of bugs. Because code is easier to generate, and multiple passes are not required, the compiler will run faster. Finally, debugging code generated by the compiler is simplified because there is no ambiguity to the debugger whether the current value of a variable is in a register or in memory.

VLSI and Upwards Compatibility

A major architectural concern in designing a VLSI microprocessor is to reduce off chip memory accesses. While switching speeds of individual transistors are increasing, the relatively constant off chip speeds will adversely affect any processor that has to make many external references. Improvements in processing technology will only make the gap wider. The reduction in memory references gained with the Stack Cache registers is therefore particularly attractive for a VLSI processor. Because the registers are invisible to the compiler, implementations with different numbers of registers create no compatibility problems; the only difference would be in the speed of the processors. As technology improves, more registers can be added without any changes to the instruction set or compiler.

A Word on Context Switching

While register saving is greatly reduced for procedure calls, registers must still be saved when a context switch occurs. Because the Stack Cache uses a larger number of registers than traditional general register machines, the time required to save them is also increased. The tradeoff being made is one of faster overall machine performance by decreasing the time for procedure calls while increasing the time for context switching. If fast context switching is extremely important, the size of the Stack Cache should be kept small or multiple register sets should be used. For the C Machine, having fast procedure calls was considered more important that having fast context switches.

Conclusion

We have described a scheme that effectively eliminates the register allocation problem by placing local variables in registers invisible to both the programmer and compiler. Registers for global variables can also be accommodated quite easily. In addition, the registers of the Stack Cache appear semantically identical to memory, eliminating the problems of aliasing and register width. Strings, arrays and structures can benefit from the improvement of register access times. The need for a large, slow, expensive and often bug ridden compiler has been eliminated while achieving most of the benefits. The hardware to implement the Stack Cache is simple, and the registers can be cheaply duplicated for parallel operand access. The mechanism described incurs minimal procedure call overhead. The technique works well because of the typical behavior of C programs, and because it is a good tradeoff between high speed hardware and the compiler technology available. The measurements clearly indicate the benefits of the caching stack entries, either with contiguous words as we have described, or with a more conventional cache memory.

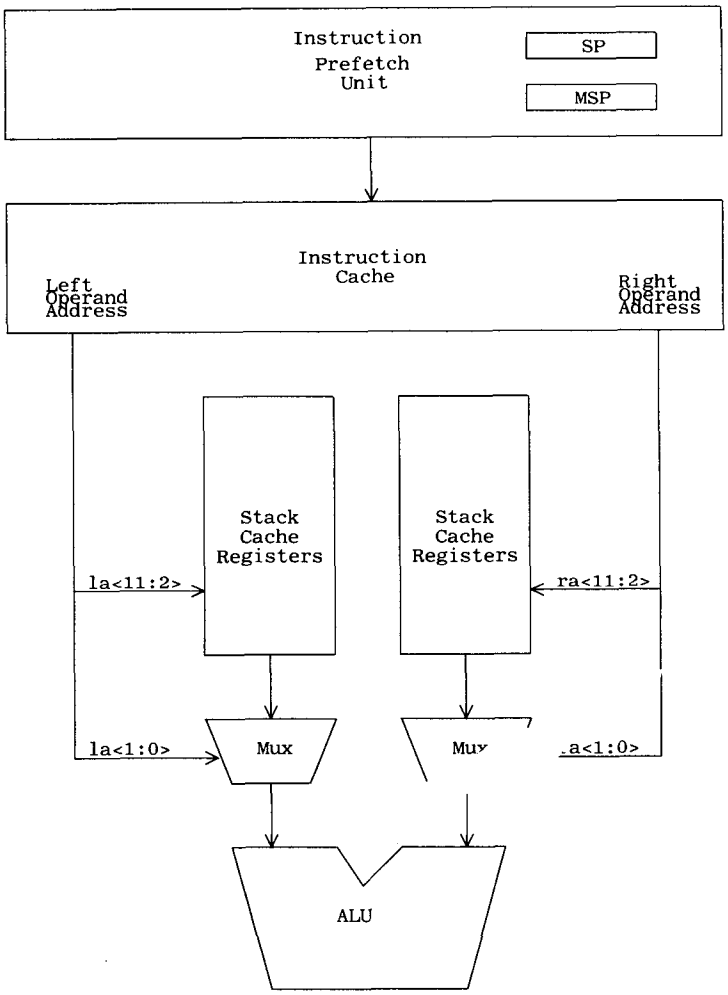
Acknowledgements

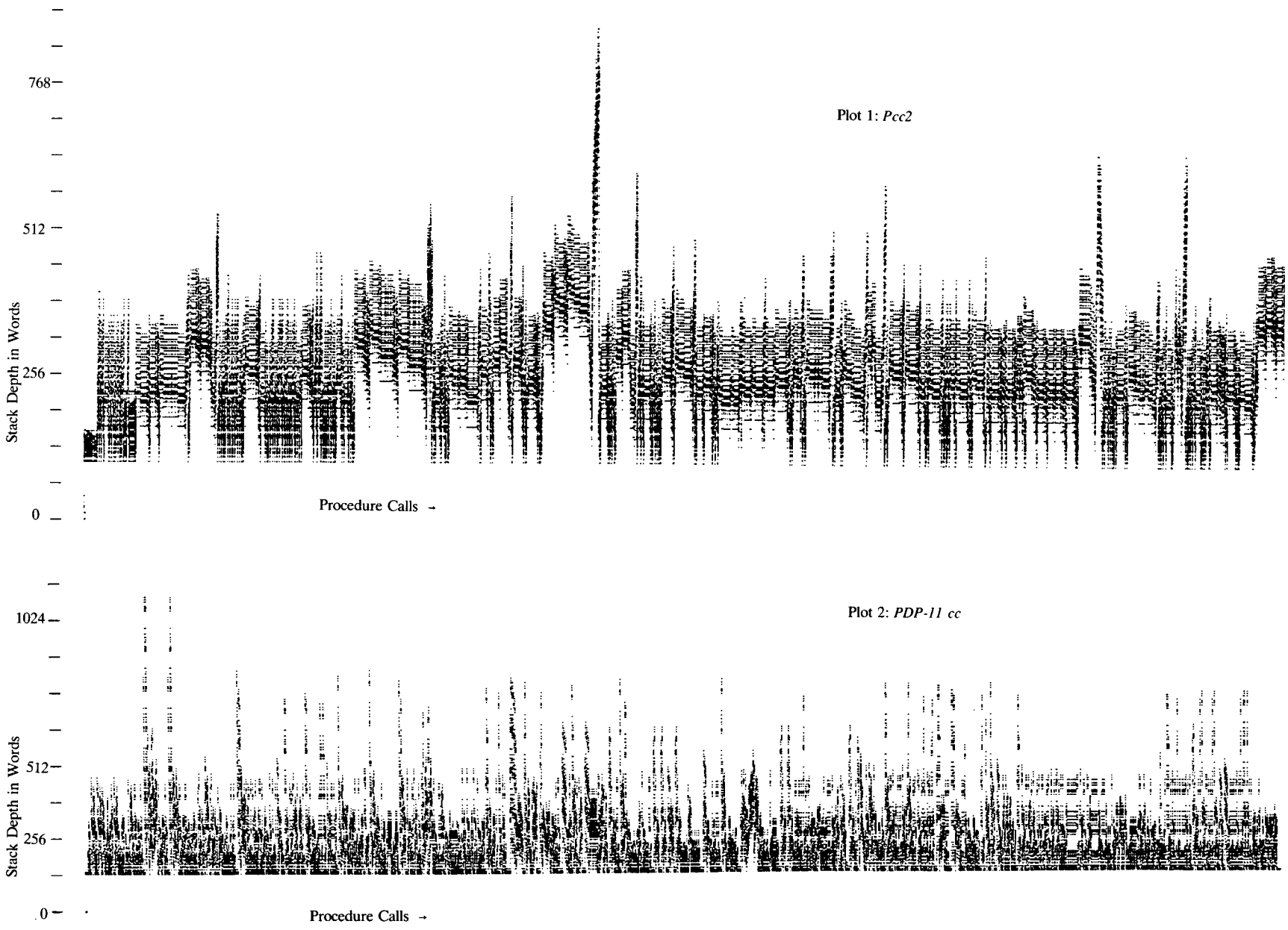
S.C. Johnson constructed the compilers and has participated in many discussions on the C Machine project. The dynamic measurements are based on work done by P.J. Weinberger. A.G. Fraser, G.L. Chesson, R.F. Cmelik and V.A. Vyssotsky are thanked for their helpful comments and support.

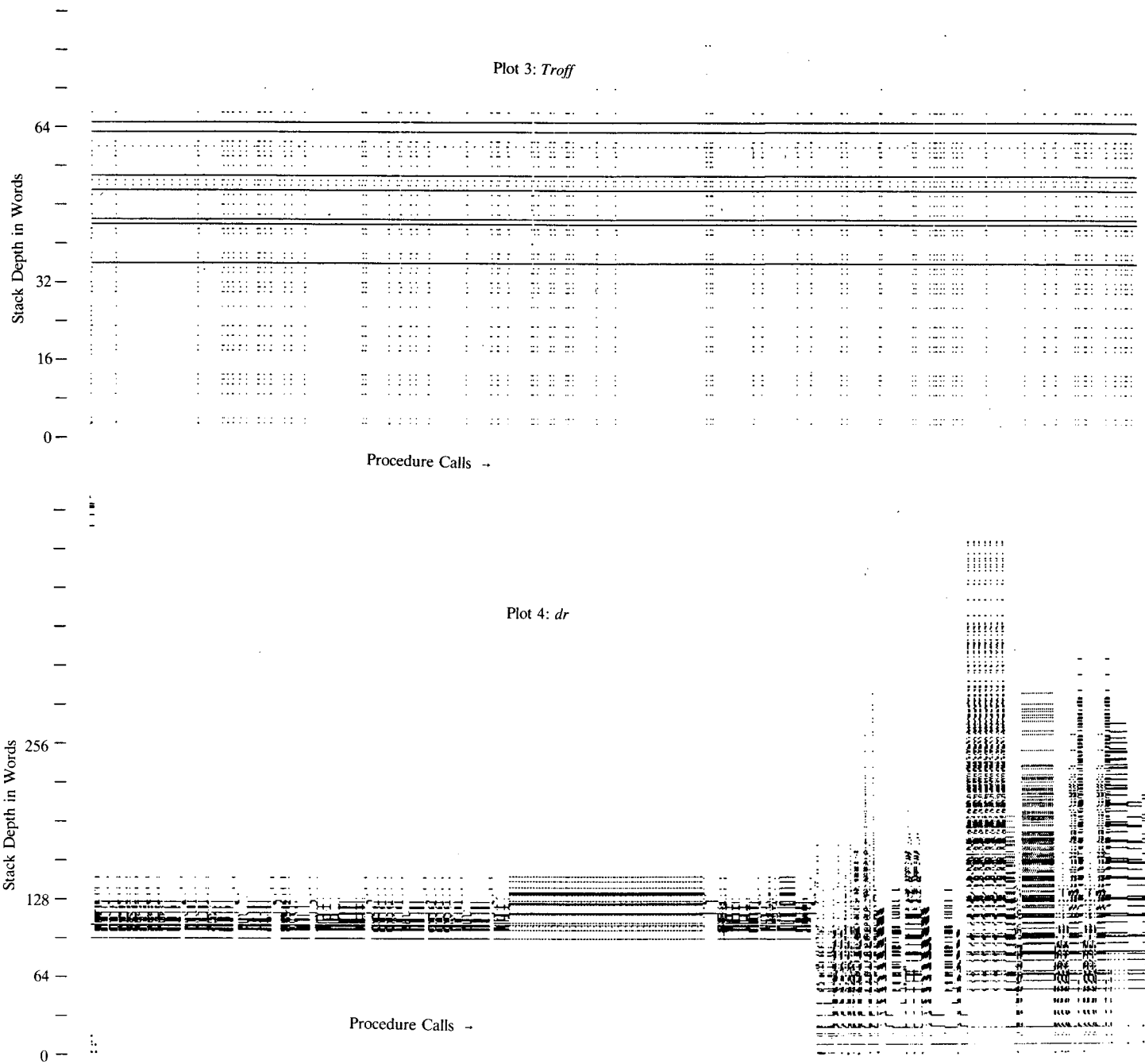
References

1. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. D.R. Ditzel, "Static and Dynamic Characteristics of the "b" Benchmarks on the VAX," Unpublished Memorandum, Bell Laboratories (July 1, 1981).
3. A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Mass. (1977).
4. R.L. Sites, "How to use 1000 Registers," *Proceedings of the Caltech Conference on VLSI*, pp. 527-532 (January, 1979).
5. R.B. Dannenberg, "An Architecture with Many Operand Registers to Efficiently Execute Block-Structured Languages," *Proceedings of the 6th Annual Symposium on Computer Architecture*, pp. 50-57 (April, 1979).
6. M. Kraley, R. Rettberg, P. Herman, R. Bressler, and A. Lake, "Design of a User-Microprogrammable Building Block," *Proceedings of the 13th Annual Microprogramming Workshop*, pp. 106-114 (December 1980).
7. *The C/70 Macroprogrammer's Handbook*, Bolt Beranek and Newman, Inc.
8. D.A. Patterson and C.H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of 8th Symposium on Computer Architecture*, pp. 443-457 (May, 1981).
9. S.T. Campbell, "MAC-8 Microprocessor Summary," in *MAC-8 Systems Designers Handbook*, Bell Laboratories (March 8, 1976).
10. D.H. Copp, "A Way to Slice the MAC-8 Register Set in Overlapping Steps," Unpublished Memorandum, Bell Laboratories (27 August, 1975).
11. G. Radin, "The 801 Minicomputer," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (March 1982).
12. D.R. Ditzel, "A Simple 32-Bit Processor for VLSI Implementation," Unpublished Memorandum, Bell Laboratories (May 1, 1981).
13. S.C. Johnson, "A 32-Bit Processor Design," Computing Science Technical Report #80, Bell Laboratories, Murray Hill, N.J. (April, 1979).
14. S.C. Johnson and D.M. Ritchie, "The C Language Calling Sequence," Computing Science Technical Report No. 102, Bell Laboratories , Murray Hill, N.J. (September 1981).
15. D.R. Ditzel and H.R. McLellan, "Measurements of C Program Stack Depth," Unpublished Memorandum, Bell Laboratories (July 24, 1981).

Figure 2. Rudiments of a 1024 word Stack Cache.




768
512
256
0
Stack Depth in Words
Procedure Calls →
Plot 1: Pcc2
1024
512
256
0
Stack Depth in Words
Procedure Calls →
Plot 2: PDP-11 cc


Plot 3: *Troff*
Stack Depth in Words
64
32
16
0
Procedure Calls →
Plot 4: *dr*
Stack Depth in Words
256
128
64
0
Procedure Calls →