THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 60, No. 2, February 1981
Printed in U.S.A.

Compiling Three-Address Code for C Programs

By J. F. REISER

(Manuscript received February 27, 1980)

This paper describes a post processor that improves the assembly-language code generated by the portable C compiler. The novel ability to change a sequence of two-address instructions into an equivalent three-address instruction distinguishes this particular code improver from other "peephole" improvers. The combined compiler-improver generates good three-address code for the Digital Equipment Corporation VAX-11® computer without requiring extensive changes in the compiler itself, which was designed to accommodate machine architectures with at most two addresses per instruction. For typical programs the improver reduces the number of bytes in the instruction stream by 10 to 23 percent. This paper emphasizes the technique used to transform two-address code to three-address code.

I. INTRODUCTION

The portable C compiler[1] is an effective tool for quickly constructing a C compiler[2] for a general purpose digital computer. With reasonable effort the resulting compiler generates correct code, and the quality of the translation into assembly language is acceptable. However, users frequently demand better code if they anticipate prolonged or extensive use of programs written for a particular application. A post processor that reads the assembly language generated by the compiler and writes better assembly language having the equivalent effect can satisfy much of the demand. (Here "better" code requires fewer bytes for instructions or less time to execute, or both.) This paper describes a program that improves code generated for the Digital Equipment Corporation VAX-11® computer, paying particular attention to the technique used to transform two-address codes into three-address codes.

One reason why a code improver can be effective is that the portable C compiler often generates code in the easiest possible correct manner, even if such a code is suboptimal over a wide range of machines. The

compiler expects that a post processor will clean up after it. For example, the compiler translates the C program fragment

```
while (···) {
        ···
    if (b > 0) break;
        ···
}
```

as if it were written

```
    while (···) {
            ···
        if (b < = 0) goto L100;
        goto L101;
L100:
            ···
    }
L101:
```

which contains a conditional jump around an unconditional jump. It should not be difficult to compile the original fragment as if it were

```
    while (···) {
            ···
        if (b > 0) goto L101;
            ···
    }
L101:
```

but the compiler does not do this, so one of the standard tasks for a code improver is to replace "skips over jumps" with jumps on the negated conditions.

Another reason that a code improver can produce better code is that the compiler's model of code generation may ignore or not take full advantage of architectural features found on a specific machine. The portable C compiler understands one-address instructions and two-address instructions, but does not understand three-address instructions or instructions which use an address as an immediate operand. Similarly, the compiler thrives on certain addressing modes (register, pointer, displacement from a register base) and has difficulty fully exploiting others (auto increment, double indexing).

A code improver can also be effective because C-language statements or compilation on a statement-by-statement basis may be too low level. The concept "turn off bit 15" may have a direct hardware implementation, but must be expressed in C language as a Boolean AND operation. The portable C compiler attempts no analysis of interstatement information flow, nor does it always take advantage of

hardware idioms. A code improver can often perform some flow analysis and recognize more hardware idioms.

The idea of a code improver is not new. "Peephole optimizers" are well known.[3,4] One C compiler for the PDP-11 computer has had a code improver for many years.* The section of Ref. 5 on the FINAL compilation pass describes a code improver used internally by a BLISS-11 compiler.

The code improver described here makes the portable C compiler usable as the workhorse compiler in a serious production environment. Measurements indicate that for typical programs the improver reduces the number of bytes in the instruction stream by 10 to 23 percent; the novel technique reported here accounts for as much as one-third of the reduction. The time required to execute the code is also reduced by 4 to 8 percent. The improver produces good three-address code from the two-address code generated by the compiler.

II. IMPROVING CODE FOR THE VAX-11

An existing improver of code compiled for the PDP-11 served as a model and outline for the VAX-11 code improver. The improver reads a file of assembly language and divides the file into segments corresponding to C procedures. For each procedure it constructs a doubly-linked list of the instructions and label definitions, with additional links for references to labels. The improver then combs the list, repeatedly trying to apply any one of several incremental transformations. The transformations satisfy a principle of optimality: Any local improvement is guaranteed to be a global improvement at least as large, and conversely, if the program as a whole can be made smaller or faster, then there is a collection of local changes which will account for the improvement. When no further transformation can be made, the improver prints the list and moves on to the next procedure. Many of the transformations depend little on the particular machine. Straightforward adaptation of the old program yielded code to transitively close jumps to jumps, delete instructions that immediately follow unconditional jumps, delete jumps to the immediately following instruction, remove unreferenced or redundant labels, merge common tail sequences, move basic blocks to the point of sole use, and interchange physical order of the consequent and alternative to a test. Simple modifications also produced a program to rotate loops to place a single conditional jump at the bottom, handle skips over jumps, eliminate redundant setting of the condition code, move common antecedents of jumps into the merged tail, eliminate constant tests or tests which are subsumed by a preceding test, exploit add-compare-

* PDP is a registered trademark of Digital Equipment Corporation.

Table I—Translations of $a = b + c$;

PDP-11	VAX-11	Improved VAX-11
mov b, r0		
add c, r0	**addl3 b, c, r0**	
mov r0, a	**movl r0, c**	**addl3 b, c, a**

branch ("DO-loop") instructions, and remember values already in registers.

III. THREE ADDRESSES FROM TWO

Fully utilizing the three-address instructions available on the VAX-11 presented a new challenge. Table I illustrates a common opportunity to use a three-address instruction. In this example the variables *a*, *b*, *c* are assumed to reside in memory (either global or local) and not in registers. The first column gives a translation for the PDP-11 that cannot be improved in either time or space. (If some of the variables reside in registers, then improvements are possible.) Both the production and the portable C compiler for the PDP-11 produce this translation without the aid of a code improver. The second column contains the code generated by the portable C compiler for the VAX-11. The compiler saves one instruction by doing the work of the first two PDP-11 instructions in one three-address VAX-11 instruction. However, it will not generate the code in the right-most column, where a single instruction suffices for the whole statement. Internally the portable C compiler uses a binary tree to represent each parsed statement. The height of a binary tree with three external nodes (each explicit variable is represented by an external node) must be at least two. Furthermore, the pattern-matching algorithms used by the compiler are restricted to subtrees of height one. (The pattern match has since been generalized to match subtrees of arbitrary height.) Thus the compiler generates two separate instructions for this case. It does have the flexibility to use an instruction with three addresses, but the destination operand of a three-address instruction must always be one of the compiler's temporary locations, usually a register. The challenge to the code improver is to recognize situations like this one and change the code appropriately.

Table II illustrates a complication. Here the addition and assignment are embedded as an expression whose value is passed as an actual argument in a procedure call. Although the same **addl3** and **movl** instructions appear together, the value in r0 is needed later and r0 cannot be elided. In standard terminology, the value in register r0 is *live*, or alternatively register r0 is *busy*. The improver can elide register usage only when the value in the register is known to be *dead*, or the register is *free*.

For an arbitrary program, determining which registers are free at a given point requires a fair amount of work. The register usage and flow of control through any part of the program can effect whether or not a register is busy in any other part of the program. Code generated by the portable C compiler has a property that makes busy/free analysis much simpler. All registers are free any time the compiler generates a backward branch instruction. The portable C compiler generates code on line, completely translating the current expression or statement before proceeding to the following expression or statement. The use of a temporary expression always occurs physically after its generation. Thus the entire busy/free analysis can be done in a single backward scan over the generated code. The backward scan marks a register busy each time the register is read or used as a source operand. Some instruction occurring closer to the front of the file must have put a live value into the register, or else the register would contain garbage. Analogously, the backward scan marks a register free each time the register is written or used as a destination operand. Since the write destroys whatever used to be in the register, no one could have wanted that dead value.

The backward scan must take precautions to record each use of a temporary register, including the implicit uses. The return instruction **ret** implicitly reads r0, the register in which C code returns function values. Thus r0 is busy just before each **ret**. The overall code-generation strategy of the compiler assumes that each procedure call instruction **calls** writes all the temporary registers. Thus all the temporary registers are free just before a procedure call.

The busy/free information can also be used to eliminate dead code. An instruction that writes only into free registers does no useful work, except possibly for the side effects it causes. If the address computations contain no side effects, then only the condition code could matter. The condition code is set by each nonbranch instruction, so the condition code itself is free unless the instruction which logically follows is a conditional branch.

The backward scan must also be careful with code generated from conditional expressions. There can be no busy registers at the time of a backward jump, as noted earlier. Since the compiler performs no

Table II—Translations of $f(a = b + c)$;

PDP-11	VAX-11	"Improved" (but wrong) VAX-11
mov b, r0		
add c, r0	**addl3 b, c, r0**	
mov r0, a	**movl r0, a**	**addl3 b, c, a**
mov r0, (sp)	**pushl r0**	**pushl r0**
jsr pc, f	**calls $1, f**	**calls $1, f**

Table III—Translation of $x = a \ ? \ b : c;$

```
        testl a
        jeql L100
        movl b, r0
        jbr L101
L100:   movl c, r0
L101:   movl r0, x
```

interstatement data-flow analysis (and in particular does not recognize common subexpressions), there can be no busy registers at the time of a forward jump generated from an entire C statement. Since labels exist only because jump instructions branch to them, these two facts might suggest that a register cannot be busy at any label, either. A register can, however, be busy at a forward jump (and thus at a label) with one of the values of a conditional expression. Table III illustrates one such situation.

Even though the instruction **movl c,r0** writes r0, the register is busy at the **jbr** because (if a is true) it contains the value of b to be stored into x. Thus the busy/free status of each register must be associated with each label as the label is passed during the backward scan, and retrieved from the corresponding label at each jump. This can be done efficiently by keeping a bit vector associated with each label, initializing all the bits to "free," and recording busy registers as labels are passed. Because backward jumps have no busy registers and the backward scan encounters the destination label of a forward jump before seeing the jump itself, the bits will always be correct.

In general the code improvements other than insertion of three-address instructions and elimination of dead code by consulting the busy/free information destroy the property that no temporary register is busy at a backward jump. This implies that using a single backward sweep over the code for the entire procedure to determine busy/free is valid only once, at the beginning before other improvements are tried. Fortunately, once is enough.

IV. OTHER USES OF THE BACKWARD SCAN

The backward prescan is also a good time to recognize hardware idioms. The VAX-11 has a number of instructions to set, clear, and test single bits, and to extract contiguous bit fields of arbitrary size. Appropriate uses of these instructions are often concealed in C with various Boolean or shift-and-mask operators or sequences of operators. Computing with the addressing modes by using instructions in which an address is used as an immediate operand often saves time and space. Powerful addressing modes often depend heavily on register usage, and the backward pass is already computing this information. Since the backward scan is performed only once, time will not be

wasted searching for hardware idioms more than once, as part of the general iterative improvement strategy. Table IV gives some example improvements.

V. DEVICE DRIVERS

On the VAX-11, the control and data registers for input/output devices lie in the memory address space. Programs manipulate the registers in much the same way as they manipulate memory, and the assembly-language code for a device driver cannot be identified solely by its form. However, certain instructions and addressing modes do not work properly when addressed to device registers. Generally these are exactly the instructions and addressing modes that the code improver wants to introduce. For example, neither of the first two improvements in Table IV is legal on a device register. Thus the code improver must be told when it is improving the code for a device driver, so it can avoid those improvements that cause problems. Reading or writing a device register typically has side effects that are different from reading or writing a memory location, and other hardware considerations such as bus widths, circuit board area, or number of words of microcode are often important. Yet from a software viewpoint such special cases are irritating and error prone, and it would be desirable to get rid of the complication.

VI. CONCLUSIONS

A single backward scan enables the code improver to determine register usage and introduce three-address instructions where appropriate. The backward scan takes advantage of the fact that all registers are free at each backward jump, a property that would otherwise be considered a weakness in the compiler. The single backward scan also recognizes hardware idioms at a lower cost than previous algorithms.

Table IV—Improvements using VAX-11 hardware idioms

C Code	Raw Translation	Improved Translation
int a; a | = 0x8000;	bisl2 $0x8000, a	jbss $15, a, L100 L100:
int a, b; b = (a ≫ 12) & 0xF;	ashl $−12, a, r0 bicl2 $−16, r0 movl r0, b	extzv $12, $4, a, b
int *p, *q; q = &p [f(x)];	pushl x calls $1, f ashl $2, r0, r0 addl2 p, r0 movl r0, q	pushl x calls $1, f moval *p [r0], q

REFERENCES

1. S. C. Johnson, "A Portable Compiler: Theory and Practice," Conference Record Fifth Ann. ACM Conf. Principles of Programming Languages, Tucson, AZ (January 1978), pp. 97–104.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall, 1978.
3. W. M. McKeeman, "Peephole Optimization," Commun. ACM, *8* (July 1965), pp. 443–4.
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Reading, Massachusetts: Addison-Wesley, 1977.
5. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, New York: Elsevier, 1975.