The Design and Application of a Retargetable Peephole Optimizer

JACK W. DAVIDSON and CHRISTOPHER W. FRASER
University of Arizona

Peephole optimizers improve object code by replacing certain sequences of instructions with better sequences. This paper describes PO, a peephole optimizer that uses a symbolic machine description to simulate pairs of adjacent instructions, replacing them, where possible, with an equivalent single instruction. As a result of this organization, PO is machine independent and can be described formally and concisely: when PO is finished, no instruction, and no pair of adjacent instructions, can be replaced with a cheaper single instruction that has the same effect. This thoroughness allows PO to relieve code generators of much case analysis; for example, they might produce only load/add-register sequences and rely on PO to, where possible, discard them in favor of add-memory, add-immediate, or increment instructions. Experiments indicate that naive code generators can give good code if used with PO.

Key Words and Phrases: code generation, optimization, peephole, portability
CR Categories: 4.12

1. INTRODUCTION

Of all optimizations, those applied to object code are among the least understood. Ad hoc instruction sets complicate formal treatment and portability. However, experience shows the value of object code optimization; even compilers with thorough global optimization reduce code size by 15–40 percent with object code optimization [12]. This is no surprise. To be machine independent, global optimization usually precedes code generation; to be simple and fast, code generators usually operate locally; so the code generator produces code fragments that can be locally optimal but may be suboptimal when juxtaposed. For example, local code for a source program conditional ends with a branch; so does local code for the end of a loop. Consequently, a conditional at the end of a loop becomes a branch to a branch. Changing the code generator to handle such situations (for instance, to generate only one branch) complicates its case analysis combinatorially, since each combination of language features may admit some optimization [6]. It is easier to simplify the code generator and to subsequently optimize object code.

A preliminary report on this research was presented at the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Tex., January 1979.
This work was supported by the National Science Foundation under Grant MCS78-02545.
Authors' address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.

Little has been published on object code optimization, and some early object code optimizations [2, 8, 9] (e.g., constant folding, exponentiation via multiplication) are now performed at a higher level [1, 11]. Recent object code optimizers [4, 12] delete unnecessary tests (a previous instruction may have incidentally set the condition code), exploit special case instructions and exotic address calculations, collapse chains of branches, delete unreachable code, and simulate register contents to replace, where possible, memory references with equivalent register references. They typically detect and correct only a few machine-specific patterns.

PO is a retargetable peephole optimizer. Given an assembly language program and a symbolic machine description, PO simulates pairs of adjacent instructions and, where possible, replaces them with an equivalent single instruction. PO makes one pass to determine the effect of each instruction, a second to collapse pairs of effects, and a third to select the cheapest instruction for each resulting effect.

There are several advantages to this organization. PO is easily retargetted by changing machine descriptions; the PDP-10, PDP-11, and Cyber 175 have already been accommodated. PO is not cluttered by ad hoc case analysis because it combines *all* possible adjacent pairs, not just branch chains or constant computations or any other special cases. As a result, PO's effect can be described formally and concisely: when PO is finished, no instruction, and no pair of adjacent instructions, can be replaced with a cheaper single instruction that has the same effect. Subsequent sections explain "adjacent" and "cheaper" and show how this thoroughness may relieve code generators of much case analysis.

The two-instruction window catches inefficiencies at the boundaries between fragments of locally generated code. It is not designed to catch others, so PO is best used with a high-level, machine-independent global optimizer.

2. MACHINE DESCRIPTIONS

To simulate an instruction, PO must know its syntax and its effect. Effects are represented as ISP register transfers [3]; for example,

R[3] ← R[3] + 1	*increments register* 3
M[c] ← 0	*clears memory location* **c**
PC ← (NZ = 0 ⇒ 140 *else* PC)	*jumps to* 140 *if* **NZ** *is zero*

PO assumes that **PC** names the program counter. No other names are assumed to have specific meanings, so PO can accommodate single-accumulator and stack machines as easily as general register machines.

A machine is described by a grammar for syntax-directed translation between assembly language and register transfers. Productions are displayed in three columns: the nonterminal being defined (in italic lowercase letters), the assembler syntax for that nonterminal (with terminal symbols in boldface uppercase letters), and the corresponding register transfer syntax. For example, these productions from a PDP-11 description

nonterminal	*assembler syntax pattern*	*register transfer pattern*
**s*	*i*	*i*
**d*	*i*	*i*
inst	**CLR** *d*	*d* ← **0**; **NZ** ← **0 ? 0**
inst	**ADD** *s*, *d*	*d* ← *d* + *s*; **NZ** ← *d* + *s* **? 0**

state that the **CLR** instruction clears its destination operand, that the **ADD** instruction adds its source to its destination, and that both set condition code bits **N** and **Z** to indicate whether the result is negative, zero, or positive. The asterisk preceding the definitions of *d* and *s* tells PO that all of their instances must match identical substrings of the register transfer. PO assumes that the program counter is automatically incremented, so this effect need not be made explicit. The productions

nonterminal	*assembler syntax pattern*	*register transfer pattern*
i	*a*	*a*
i	@*a*	**M**[*a*]

state that a word operand *a* may be preceded by a "@" for an indirect reference. Finally, the productions

nonterminal	*assembler syntax pattern*	*register transfer pattern*
a	*x*	**M**[*x*]
a	#*x*	*x*
a	**R***n*	**R**[*n*]
a	*x*(**R***n*)	**M**[**R**[*n*]+*x*]

define how the word operand *a* may appear in assembly code: the lone address *x* stands for the named memory location **M**[*x*], #*x* stands for the literal value *x*, **R***n* stands for the named register **R**[*n*], and *x*(**R***n*) stands for an indexed address where address *x* is the base and register **R**[*n*] holds the offset. The primitive nonterminals are *x*, which stands for a symbolic address, and *n*, which stands for a register index. Appendix A contains additional descriptions for the PDP-11.

Details irrelevant to the object code may be omitted from the machine description. For example, PO does not need to know how the condition code represents comparisons, so the machine description does not say. Similarly, instructions that PO has little or no chance of collapsing (e.g., **HALT**, block moves, subroutine calls, and returns) may be omitted. Undescribed instructions may appear in programs—PO will not disturb them.

The machine description may disguise awkward machine features. For example, PDP-11 conditional branches can only reach nearby words; assemblers—and PO machine descriptions based on them—disguise this fact by allowing conditional branches to arbitrary targets and translating them into two-instruction sequences when "short" branches cannot reach. Similarly, the PDP-11 hardware does not offer immediate operands; instead it offers "autoincrement" addressing, which references indirectly through an index register that is subsequently incremented. Since the program counter is also an index register, assemblers—and PO machine descriptions—offer immediate operands by generating the less obvious autoincrement through the program counter [10].

Since PO knows target machines only through these patterns, it is retargetted by describing a different instruction set. Its few machine dependencies are assumptions built into its algorithms and machine description language. For example, the simulator assumes that the machine uses a program counter and that cells, once set, stay set: PO cannot optimize code that uses changing device registers. Similarly, PO's machine descriptions cannot easily represent instructions with internal loops (e.g., block moves, which may appear in programs but will not be collapsed). In general, such assumptions can be removed by extending PO. As it stands, PO can handle most instructions on most machines.

3. DETERMINING EFFECTS

PO needs to know the effect—in register transfers—of each instruction. If PO is built into a compiler, the code generator can emit register transfers equivalent to the object code instructions that it would otherwise generate, and PO can proceed directly with collapsing pairs of instructions. On the other hand, if PO is to accept assembly code, it must first make a pass to determine the effect of each assembler statement in isolation (so PO assumes that programs do not modify themselves). Given an assembler statement, it seeks a matching assembler syntax pattern and returns the corresponding register transfer pattern, with pattern variables evaluated. For example, the instruction

ADD #2, R3

matches the syntax pattern

ADD *s*, *d*

so PO substitutes **2** (the translation of **#2**) for *s* and **R[3]** (the translation of **R3**) for *d* in the register transfer pattern

$d \leftarrow d + s$; **NZ** $\leftarrow d{+}s$ **? 0**

and obtains

R[3] ← R[3] + 2; NZ ← R[3]+2 ? 0

Programs typically ignore some effects of some instructions. For example, a chain of arithmetic instructions may set and reset condition codes without ever testing them. PO can do a better job if such useless register transfers are removed from an instruction's register transfer list. For example, the full effect of the instruction above includes assignments to the **N** and **Z** bits. If the next instruction changes **N** and **Z** without testing them, its useful effect is only

R[3] ← R[3] + 2

If the previous instruction references **R[3]** indirectly, the useful effect may be had by autoincrementing instead and removing the **ADD** instruction. The full effect requires the **ADD** instruction, since autoincrementing does not set the condition code. Consequently, when initially determining each instruction's effect, PO ignores effects on such "dead" variables. To do so, the initial pass scans the program backward and associates with each instruction both its useful effect and a list of cells that are unused and therefore may be changed arbitrarily. Each instruction's list is computed to be that of its lexical successor, plus the cells its successor sets, minus the cells its successor examines. If the instruction branches, its list is taken to be empty, since the dead variables depend on the destination of the branch. Full dead variable elimination (considering control flow and subscripting) [7] is an unnecessary expense, for this simpler analysis permits the first pass over the code to eliminate most "extra" effects such as condition code setting. As a bonus, code not subjected to dead variable elimination at a higher level enjoys a measure of it now: instructions with no effect are removed. If PO is used with a code generator that produces register transfers instead of assembly code, the code generator will not produce extra effects and can report when temporaries become dead, so a pass for dead variable identification should not be needed.

4. COLLAPSING PAIRS

Once PO knows the isolated effect of each instruction, it passes forward over the program and considers the combined effect of lexically adjacent instructions; where possible, it replaces such pairs with a single instruction having the same effect. PO learns the effect of a pair by combining their independent effects and substituting the values assigned to variables in the first for instances of those variables in the second. The effect of

SUB #2, R3
CLR @R3

is (ignoring dead variable elimination)

R[3] ← R[3] − 2; NZ ← R[3]−2 ? 0
M[R[3]] ← 0; NZ ← 0 ? 0

which simplifies to

R[3] ← R[3] − 2; M[R[3] − 2] ← 0; NZ ← 0 ? 0

PO now seeks a single instruction with a register transfer pattern matching this effect. It finds the autodecrement version of **CLR**

CLR −(R3)

A register transfer pattern matches if it performs all register transfers requested and if the rest of its register transfers set harmless dead variables (e.g., the condition code). After each replacement, PO backs up one instruction—to consider the new adjacency between the new instruction and its predecessor—and continues.

Pairs that start with a branch need special treatment. The condition on which the branch depends must be inverted and added to the register transfers of the second instruction before combining effects. For example, the two instructions

PC ← (NZ = 0 ⇒ l_1 *else* PC)
PC ← l_2
l_1:

combine to

PC ← (NZ = 0 ⇒ l_1 *else* PC); PC ← (*not* NZ = 0 ⇒ l_2 *else* PC)
l_1:

A symbolic simplifier now improves awkward relationals and deletes redundant PC assignments, yielding

PC ← (NZ ≠ 0 ⇒ l_2 *else* PC)
l_1:

for the example above. Unconditional branches depend on the constant condition ***true***; the symbolic simplifier deletes register transfers depending on its inverse, removing unreachable code.

Labels prevent the consideration of some pairs. Combining pairs whose second instruction is labeled changes, erroneously, the effect of programs that jump to the label to include the effect of the first instruction. PO must ignore such pairs and assume that all branches are to explicit labels. To improve its chances, PO removes any labels it can. When it encounters a label, it looks for a reference to it; if it finds none—possibly because optimizations like the one above have removed them all—PO removes the label and tries combining the two instructions that it separated. This technique enables PO to remove the last three branches in the large example below.

When PO removes the last reference to a label that it has passed, it should back up to reconsider the instructions the label separated: new optimizations are possible after the label is removed. This reconsideration is needed only for labels referenced following their definition. However, when optimizing the code generated locally from a program with "structured" control flow, loop and subroutine heads are the only such labels, and peephole optimizers seldom remove these labels. So this particular form of backup, though easily implemented and theoretically necessary, was discarded as ineffective.

PO collapses branch chains by treating a branch and its target as an extra pair. If an instruction branches to l, PO concatenates the branch instruction with the instruction at l, attempts optimization of this pair, and replaces the first branch (leaving the instruction at l alone) if possible. For example, the PDP-10 sequence

$$\begin{array}{ll} & \mathbf{JRST}\ l_1 \\ & \ldots \\ l_1: & \mathbf{AOJG}\ 3, l_2 \end{array}$$

has the effect

$$\begin{array}{ll} & \mathbf{PC} \leftarrow l_1 \\ & \ldots \\ l_1: & \mathbf{R[3]} \leftarrow \mathbf{R[3]} + 1;\ \mathbf{PC} \leftarrow (\mathbf{R[3]}+1 > 0 \Rightarrow l_2\ \textit{else}\ \mathbf{PC}) \end{array}$$

which combines to

$$\begin{array}{ll} & \mathbf{R[3]} \leftarrow \mathbf{R[3]} + 1;\ \mathbf{PC} \leftarrow (\mathbf{R[3]}+1 > 0 \Rightarrow l_2\ \textit{else}\ \mathbf{PC}) \\ & \ldots \\ l_1: & \mathbf{R[3]} \leftarrow \mathbf{R[3]} + 1;\ \mathbf{PC} \leftarrow (\mathbf{R[3]}+1 > 0 \Rightarrow l_2\ \textit{else}\ \mathbf{PC}) \end{array}$$

so PO replaces the first instruction with the second. (Note that the second instruction may now be unreachable.) PO does not make the replacement if it requires the introduction of a new label. For example, if the second instruction had the effect

$$l_1:\ \mathbf{R[3]} \leftarrow \mathbf{R[3]} + 1$$

PO *could* replace the first instruction with

$$\mathbf{AOJG}\ 3{,}l_1{+}1$$

but it does not because, as shown above, introducing new labels (l_1+1) prevents the consideration of other pairs. PO combines only physically adjacent instructions and branch chains.

When this pass reaches the end of the program, PO makes a third and final pass to translate the remaining register transfers back to assembly code. When searching for an instruction that realizes a particular set of register transfers, PO scans the instruction list in order, so cheaper instructions should be described first. Occasionally two (or more) instructions are better than one; the general solution to this problem is to add instruction timings to machine descriptions, but it is less expensive and just as practical to describe the two-instruction sequence as a macro instruction and place it before the less desirable single instruction. This third pass could be absorbed into the second pass if the second pass kept track of register transfers *and* the equivalent assembly code.

For purposes of comparison, Appendixes B–D show PO optimizing a 30-instruction program that has been used to illustrate FINAL, the PDP-11-dependent object code optimizer of the BLISS-11 optimizing compiler [12]. PO yields 19 instructions; by simply combining adjacent instructions, it collects branch chains, uses special-purpose addressing modes, combines jumps-over-jumps, and deletes useless tests and unreachable code. FINAL yields 16 instructions, because it does "cross-jumping," a reordering that can eliminate redundant code. Cross-jumping may permit other optimizations but, by itself, does not make programs faster, only smaller. Hence, it differs fundamentally from PO's optimizations; even a wider window would not help. Cross-jumping could be added to PO, but the larger need is for a space-optimizer that reduces code size through general reorderings.

5. CODE GENERATION

PO can greatly reduce the number of cases that a code generator must consider to produce quality code. Suppose that the early, largely machine-independent compiler phases produce intermediate postfix code for a simple stack machine. For example,

i ← i − 1

might be translated to

PUSH i	*push the address* **i**
INDIR	*replace it with the word it addresses*
PUSH 1	*push the constant* **1**
SUB	*subtract the* **1** *from* **i**
PUSH i	*push the address* **i**
STORE	*store the result in* **i**

Though bulky, this code is easy to generate. Furthermore, it is easy to write macros that expand it into code for target machine. For example, the macros might rewrite the code above in PDP-11 assembly language, simulating the stack in words **a** and **b**:

MOV #i,a	*move address* **i** *to* **a**
MOV @a,a	*replace it with the word it addresses*

Table I

Function	Machine	Postfix	Before PO	After PO	Host compiler	Hand code
tprint	PDP-11	73	76	19	16	16
ctoi	PDP-10	56	60	28	20	18
ctoi	Cyber 175	56	64	41	38	26
mmult	PDP-11	81	95	41	40	22
mmult	PDP-10	81	84	39	26	19
mmult	Cyber 175	81	93	69	61	27

MOV #1,b	*move* **1** *to* **b**
SUB b,a	*subtract* **b** *from* **a**
MOV #i,b	*move address* **i** *to* **b**
MOV a,@b	*move* **a** *to the word addressed by* **b**

The macros need know only the most general instructions (e.g., subtract, not decrement) and the most primitive addressing modes (enough to fetch addresses and simulate a stack) because PO can introduce better ones. For example, PO first reduces each of the three pairs above, yielding

MOV i,a	*move* **i** *to* **a**
DEC a	*decrement* **a**
MOV a,i	*move* **a** *to* **i**

Then it uses a three-instruction window to reduce these to the optimal

DEC i	*decrement* **i**

PO needs the larger window when working with naive code generators because, while many machines offer some one-instruction replacements for load/operate/store sequences, few offer replacements for the load/operate and operate/store subsequences; PO must look at all three to reduce them to one. Checking triples slows PO but does not make it much more complex because it uses the pair-handling machinery to combine triples. Fortunately, no special need has been observed for a still larger window or a more complex replacement strategy (e.g., replacing triples with equivalent pairs).

Table I shows how this strategy has performed on larger examples. The numbers give the sizes (in instructions) of the stack machine code, the target machine code before and after optimization, and similar code produced by a more conventional, machine-dependent optimizing compiler and by an assembly language expert for three subroutines: **tprint** prints trees, **ctoi** converts strings to integers, and **mmult** multiplies matrices.

In fact, PO can come even closer: in general, the host compilers did better, not because of superior case analysis during code generation, but because they assign crucial variables to registers and perform global optimizations. Because such improvements are largely machine independent, they could be added to PO's compiler without making it much harder to retarget.

6. DISCUSSION

PO is a five-page SNOBOL program that runs in 128K bytes on a PDP-11/70; the program includes the simple one-page code generator outlined in the last section.

It uses a two-page preprocessor to turn machine descriptions into SNOBOL patterns. Machine descriptions are about two pages and can be written in an hour or two by someone who knows the machine. This version trades speed for simplicity; for example, it will look to see if a register transfer matches a decrement pattern even if it already has failed to match a more general subtract pattern. Such shortcuts slow PO: it typically processes only 1–10 instructions each second, and this rate changes linearly with the number of patterns in the machine description. The design of a production version is underway; for example, it uses a table-driven pattern matcher that dismisses decrements when it dismisses subtracts and is relatively insensitive to the size of the machine description. Preliminary experiments indicate that this version will run fast enough for everyday use, though conventional, hand-coded peephole optimizers will probably remain faster.

PO's relative lack of context is also being addressed. Repeated application of two- and three-instruction windows can increase the effective window size (witness the reduction of six instructions to one above), but sometimes more context is needed. For example, PO cannot collapse an otherwise-reducible pair separated by a third, uncombinable instruction; hand-coded peephole optimizers can. The production version of PO may use simple data flow analysis to identify such nonadjacent pairs that are likely candidates for combining.

Experience with PO also suggests reexamining the division of labor between the global optimizer, register allocator, code generator, and object code optimizer. PO eliminates (much) unreachable code; perhaps global optimizers should not bother with this improvement. A recent code generator [5] matches intermediate code against machine description patterns to guide local register allocation; since PO does similar matching, perhaps it can allocate registers. Register transfers resemble the quadruples that many global optimizers use to represent programs; perhaps a machine-independent, global optimizer [7] can be adapted to accept a machine description and use its more global view of object code to catch inefficiencies missed by PO's narrow window.

APPENDIX A. PDP-11 DESCRIPTION

This is about 40 percent of the PDP-11 machine description; only primitive nonterminals, a few instructions, and some other uninteresting details have been omitted.

nonterminal	*assembler syntax pattern*	*register transfer pattern*
a	**R*n***	**R[*n*]**
a	**(R*n*)+**	**M[R[*n*]++]**
a	**−(R*n*)**	**M[−−R[*n*]]**
a	***x*(R*n*)**	**M[R[*n*]+*x*]**
a	*x*	**M[*x*]**
a	#*x*	*x*
i	*a*	*a*
i	**@*a***	**M[*a*]**
**d*	*i*	*i*
**s*	*i*	*i*
inst	**TST *d***	**NZ←*d*?0;**
inst	**CMP *s*,*d***	**NZ←*s*?*d*;**
inst	**CLR *d***	***d*←0; NZ←0?0;**

nonterminal	*assembler syntax pattern*	*register transfer pattern*
inst	**MOV** *s,d*	*d*←*s*; NZ←*s*?0;
inst	**INC** *d*	*d*←*d*+1; NZ←*d*+1?0;
inst	**DEC** *d*	*d*←*d*−1; NZ←*d*−1?0;
inst	**ASL** *d*	*d*←*d**2; NZ←*d**2?0;
inst	**ASR** *d*	*d*←*d*/2; NZ←*d*/2?0;
inst	**ADD** *s,d*	*d*←*d*+*s*; NZ←*d*+*s*?0;
inst	**SUB** *s,d*	*d*←*d*−*s*; NZ←*d*−*s*?0;
inst	**BR** *a*	PC←*a*;
inst	**B***rel a*	PC←NZ *rel* 0 ⇒ *a* ***else*** PC;

APPENDIX B. TREE PRINTER[1]

This is the PDP-11 assembly code produced by the first phases of the BLISS-11 compiler for a program that prints trees. In addition to the effects shown, each nonbranch sets the condition code according the value it assigns; TSTs set the condition code but do nothing else.

		assembly code		*effect*
(1)		JSR	R1, sav3	call sav3
(2)		MOV	S+310,R3	R[3] ← M[S+310]
(3)		MOV	12(R5),R2	R[2] ← M[R[5]+12]
(4)		ADD	#177776,R3	R[3] ← R[3] − 2
(5)		CLR	@R3	M[R[3]] ← 0
(6)	l_5:l_6:	TST	left(R2)	NZ ← M[R[2]+left] ? 0
(7)		BNE	l_7	PC ← NZ ≠ 0 ⇒ l_7 *else* PC
(8)		BR	l_8	PC ← l_8
(9)	l_7:	ADD	#177776,R3	R[3] ← R[3] − 2
(10)		MOV	R2,@R3	M[R[3]] ← R[2]
(11)		MOV	left(R2),R2	R[2] ← M[R[2]+left]
(12)		BR	l_6	PC ← l_6
(13)	l_8:	MOV	info(R2),R1	R[1] ← M[R[2]+info]
(14)		JSR	R7,print	call print
(15)	l_9:	MOV	right(R2),R2	R[2] ← M[R[2]+right]
(16)		TST	R2	NZ ← R[2] ? 0
(17)		BEQ	l_{10}	PC ← NZ = 0 ⇒ l_{10} *else* PC
(18)		BR	l_{11}	PC ← l_{11}
(19)	l_{10}:	MOV	@R3,R2	R[2] ← M[R[3]]
(20)		ADD	#2,R3	R[3] ← R[3] + 2
(21)		TST	R2	NZ ← R[2] ? 0
(22)		BNE	l_{12}	PC ← NZ ≠ 0 ⇒ l_{12} *else* PC
(23)		BR	l_{13}	PC ← l_{13}
(24)	l_{12}:	MOV	info(R2),R1	R[1] ← M[R[2]+info]
(25)		JSR	R7,print	call print
(26)		BR	l_{14}	PC ← l_{14}
(27)	l_{13}:	BR	l_4	PC ← l_4
(28)	l_{14}:	BR	l_9	PC ← l_9
(29)	l_{11}:	BR	l_5	PC ← l_5
(30)	l_4:	RTS	R7	return

[1] With thanks to Elsevier Publishing.

APPENDIX C. PO'S PAIRWISE OPTIMIZATIONS ON TREE PRINTER

In most cases, PO replaces the pair named with one equivalent instruction; comments note the three reductions involving nonadjacent branch chain members where the second instruction is retained.

	pair	*result instruction*			*explanation*
(a)	4,5		**CLR**	**−(R3)**	use autodecrement
(b)	7,8		**BEQ**	**l_8**	remove label l_7
(c)	9,10		**MOV**	**R2,−(R3)**	use autodecrement
(d)	15,16	l_9:	**MOV**	**right(R2),R2**	remove **TST**
(e)	17,18		**BNE**	**l_{11}**	remove label l_{10}
(f)	(e),29		**BNE**	**l_5**	remove label l_{11}, retain 29
(g)	19,20		**MOV**	**(R3)+,R2**	use autoincrement
(h)	(g),21		**MOV**	**(R3)+,R2**	remove **TST**
(i)	22,23		**BEQ**	**l_{13}**	remove label l_{12}
(j)	(i),27		**BEQ**	**l_4**	remove label l_{13}, retain 27
(k)	26,27		**BR**	**l_{14}**	27 unreachable without l_{13}
(l)	(k),28		**BR**	**l_9**	remove label l_{14}, retain 28
(m)	(l),28		**BR**	**l_9**	28 unreachable without l_{14}
(n)	(m),29		**BR**	**l_9**	29 unreachable without l_{11}

APPENDIX D. OPTIMIZED TREE PRINTER

Here is the tree printer after PO's optimizations. FINAL's cross-jumping optimization changes the last branch to go to l_8 instead of l_9 and eliminates the second **MOV/JSR** sequence. This, in turn, allows it one last optimization: the now-adjacent **BEQ** and **BR** can be combined into a **BNE.**

		assembly code		*effect*
(1)		**JSR**	**R1,sav3**	**call sav3**
(2)		**MOV**	**S+310,R3**	**$R[3] \leftarrow M[S+310]$**
(3)		**MOV**	**12(R5),R2**	**$R[2] \leftarrow M[R[5]+12]$**
(4)		**CLR**	**−(R3)**	**$M[R[3]-2] \leftarrow 0$; decrement R[3]**
(6)	l_5:l_6:	**TST**	**left(R2)**	**$NZ \leftarrow M[R[2]+left]$? 0**
(7)		**BEQ**	**l_8**	**$PC \leftarrow NZ = 0 \Rightarrow l_8$ *else* PC**
(9)		**MOV**	**R2,−(R3)**	**$M[R[3]-2] \leftarrow R[2]$; decrement R[3]**
(11)		**MOV**	**left(R2),R2**	**$R[2] \leftarrow M[R[2]+left]$**
(12)		**BR**	**l_6**	**$PC \leftarrow l_6$**
(13)	l_8:	**MOV**	**info(R2),R1**	**$R[1] \leftarrow M[R[2]+info]$**
(14)		**JSR**	**R7,print**	**call print**
(15)	l_9:	**MOV**	**right(R2),R2**	**$R[2] \leftarrow M[R[2]+right]$**
(17)		**BNE**	**l_5**	**$PC \leftarrow NZ \neq 0 \Rightarrow l_5$ *else* PC**
(19)		**MOV**	**(R3)+,R2**	**$R[2] \leftarrow M[R[3]]$; increment R[3]**
(22)		**BEQ**	**l_4**	**$PC \leftarrow NZ = 0 \Rightarrow l_4$ *else* PC**
(24)		**MOV**	**info(R2),R1**	**$R[1] \leftarrow M[R[2]+info]$**
(25)		**JSR**	**R7,print**	**call print**
(26)		**BR**	**l_9**	**$PC \leftarrow l_9$**
(30)	l_4:	**RTS**	**R7**	**return**

REFERENCES

1. ALLEN, F.E., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, no. 1–30, R. Rustin (Ed)., Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. BAGWELL, J.T. Local optimizations. SIGPLAN Notices *5*, 7 (July 1970), 52–66.
3. BELL, C.G., AND NEWELL, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.
4. FORTRAN-10 reference manual. Digital Equipment Corp., Maynard, Mass., 1974.
5. GLANVILLE, S., AND GRAHAM, S.L. A new method for compiler code generation. In Conf. Rec. 5th Annu. ACM Symp. Principles of Programming Languages, 1978, pp. 231–240.
6. HARRISON, W. A new strategy for code generation—The general purpose optimizing compiler. In Conf. Rec. 4th ACM Symp. Principles of Programming Languages, 1977, pp. 29–37.
7. HECHT, M.S. *Flow Analysis of Computer Programs.* North-Holland, Amsterdam, 1977.
8. LOWRY, E.S., AND MEDLOCK, C.W. Object code optimization. *Commun. ACM 12*, 1 (Jan. 1969), 13–22.
9. MCKEEMAN, W.M. Peephole optimization. *Commun. ACM 8*, 7 (July 1965), 443–444.
10. PDP-11 processor handbook. Digital Equipment Corp., Maynard, Mass., 1975.
11. STANDISH, T.A., HARRIMAN, D.C., KIBLER, D.F., AND NEIGHBORS, J.M. The Irvine program transformation catalogue. Dep. Information and Computer Science, Univ. California, Irvine, 1976.
12. WULF, W., JOHNSSON, R.K., WEINSTOCK, C.B., HOBBS, S.O., AND GESCHKE, C.M. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

Received July 1979; revised December 1979; accepted January 1980