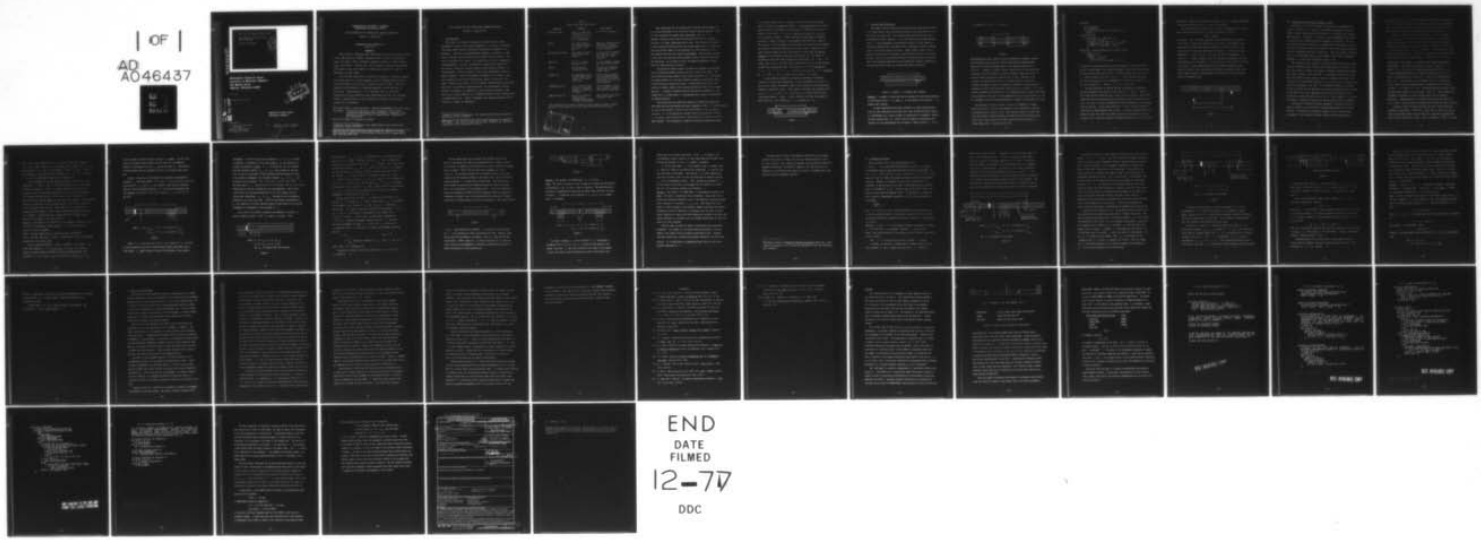
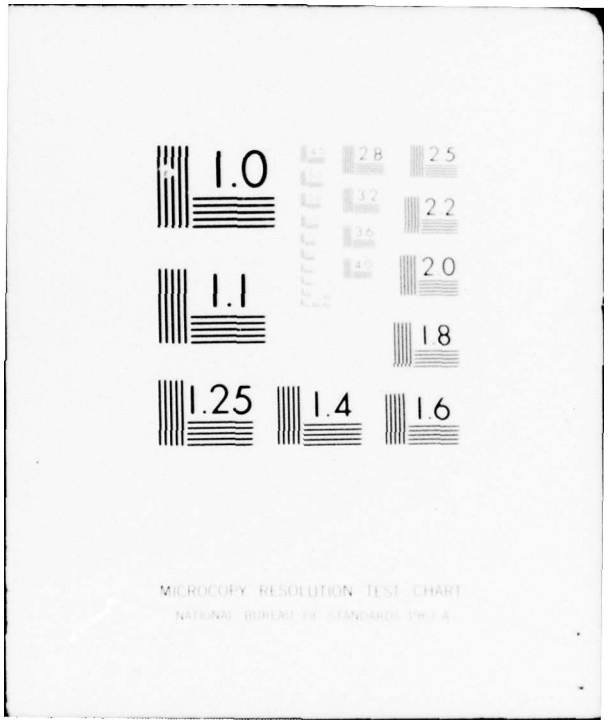
AD-A046 437 WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
CODE GENERATION FOR SHORT/LONG ADDRESS MACHINES.(U)
AUG 77 E L ROBERTSON DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-1779 NL

| OF |
AD
A046437



END
DATE
FILMED
12-77
DDC


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

MRC Technical Summary Report # 1779

CODE GENERATION FOR SHORT/LONG ADDRESS MACHINES

Edward L. Robertson

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

August 1977

(Received June 30, 1977)

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D. C. 20550

(See 1473)

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

CODE GENERATION FOR SHORT/LONG ADDRESS MACHINES

Edward L. Robertson*

Technical Summary Report # 1779
August 1977

ABSTRACT

Many machine languages have different instruction formats which allow addressing of "nearby" operands with a "short" instruction (e.g. one word), while "faraway" operands require a "long" format (e.g. two words). Because the size of object code may depend upon the formats used, the formats of different instructions may be dependent on each other.

An efficient algorithm is given for optimally assigning formats to instructions in a given program, and an implementation will be discussed which is practical in space as well as time. The more sophisticated problem of arranging operands within programs is discussed. Unfortunately, it is unlikely that an efficient algorithm can even guarantee good approximations for this problem. Finally, implications of this problem on hardware and software designs are considered.

AMS (MOS) Subject Classification - 68A10; CR Categories - 4.11, 4.12, 5.25

Key Words: short/long addresses, span-dependent instructions, variable-length addressing, assembler, compiler, code-generation, optimization, computational complexity, NP-completeness

Work Unit Number 8 (Computer Science)

* Computer Science Department, The Pennsylvania State University, University Park, PA 16802.

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.
The research reported in this document was partially supported by NSF Grant MCS-76-17323.

CODE GENERATION FOR SHORT/LONG ADDRESS MACHINES

Edward L. Robertson*

I. Introduction

One aspect of machine organization which occurs quite frequently in mini- and micro-computers is a variety of addressing modes which require different instruction lengths in one machines. The most straightforward example is perhaps in IBM 1130, in which instructions may be either "short-format", ±127 words (actually +127 words, -128 words due to two's-complement representation - we will occasionally ignore this detail in the future) from the current value of the program counter, or "long-format", with an entire 16 bit word containing a direct memory address. Thus a reference or branch to a "nearby" location requires a one-word instruction, while a reference "far-away" requires two words. If we consider housekeeping operations necessary to establish addressability (of information already present in memory, as opposed to, say, paging overhead), then even such a large machine as an IBM/370 exhibits these differences in addressing cost. Table I compares the addressing mode variations of a number of machines.

* Computer Science Department, The Pennsylvania State University, University Park, PA 16802.

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024. The research reported in this document was partially supported by NSF Grant MCS-76-17323.

TABLE I:

Addressing mode variations

machine	"nearby"	"far-away"
Motorola 6800	bytes 0 thru 255 ("direct"), PC±127 ("relative"), or index register + 8-bit displacement	16 bit byte address
PDP-8	128 word "pages", can address page 0 or current (PC) page	Indirect addressing of 4096 word "fields," 3 bit field pointers ("very faraway")
Microdata 1600/30	Can address page 0 (256 words) or PC±127	15 bit direct address, can generate 16 bit with use of index register
IBM 1130	PC±127 or index register ±127	16 bit address (indexing and indirection)
PDP-11	PC±127 words (*BRANCHES ONLY*)	16 bit byte address (indexing, indirections, etc.)
PRIME 400	Displacement of up to 255 words from stack or PC base registers	16 bit displacement from procedure base, also various indirections (16, 32, or 48-bit pointers)
INTERDATA 8/32	14 bit direct and 15 bit PC-relative address (indexing allowed)	24 bit address (8/32 uses 20 bits; single and double indexing)
IBM/360 and 370	Up to 15 (usually fewer) base register pointing to 4096-byte blocks	Must load a new base register value

(PC indicates the current value of the program counter, which points to the location of the instruction being executed.)

When generating code for machines with different address modes, it is of course desirable to use the shorter formats whenever possible. This not only decreases the program space requirement, but may also decrease execution time, since an additional fetch cycle is usually necessary if a longer addressing mode is used. This concern is therefore a code optimization problem, which may be approached either in a basic way, using the minimal number of long addresses in a fixed program, or in a more sophisticated way with program rearrangement. The principle activity in the later case would be moving variables "nearby" locations where they are referenced. As is often the case, the simple optimization is easy, while the sophisticated one is hard.

The particular model we adopt--of course the simplest one available--has two addressing formats, which we will call "short" and "long". All instructions are single address, with a short instruction (i.e. an instruction whose address is in the short format) occupies one word (unit) while a long instruction occupies two words. A long address may refer to any location in memory, while a short instruction may refer only up to a fixed distance ℓ forward or backward from the instruction containing it. Although this is a simple model, it corresponds quite closely to an IBM 1130 (as described above).

The notation for this model will consist of identifying labels with data words and with instructions and their operands, so that $\alpha:\beta$ will represent an instruction labelled α which has the word labelled β as an operand. Note that $\alpha:\beta$ is an affirmative statement about the instruction labelled α. Of course, each instruction has a unique label but a label may occur in many operands. The occurrences of labels on instructions defines an order

(<) on these labels (which is a partial to the extent that the program consists of position-independent modules). The distance between two instructions on labels is the number of words between the instructions (a word is distance 0 from itself, +1 from its immediate predecessor or successor). If an assembler, compiler, or other code-modifying procedure is changing instruction formats, distance is then dependent upon the time or stage of the procedure, since the number of words between two instructions may depend upon whether intervening instructions are long or short. However, we will always begin presuming that all instructions are short; thus the initial distance will be defined to take this into account, i.e., the number of instructions (plus data words) between two instructions. If α and β are labels, then $di(\alpha,\beta)$ is their initial distance and $d(\alpha,\beta)$ is the distance at some (here unspecified) time. An instruction $\alpha:\beta$ is in the span of $\gamma:\delta$ if $\gamma < \alpha < \delta$ or $\delta < \alpha < \gamma$. In this case we also say $\gamma:\delta$ is dependent on $\alpha:\beta$ - that is, $d(\gamma,\delta)$ changes if the format of $\alpha:\beta$ does.

A simple graphical notation is often more convenient to represent structures of labels and instructions. We will draw memory as a series of cells across a page, with address labels written above the cells and operand addresses indicated by arrows. Thus, Figure 1 represents the following facts: $\alpha:\beta$, $\beta:\gamma$, $\gamma < \alpha < \beta$, $d(\alpha,\gamma) = 1$, and $d(\beta,\gamma) = k$. Observe that the words intervening between β and γ are not indicated, but they are assumed to have fixed size. The instruction $\beta:\gamma$ is dependent on $\alpha:\beta$.

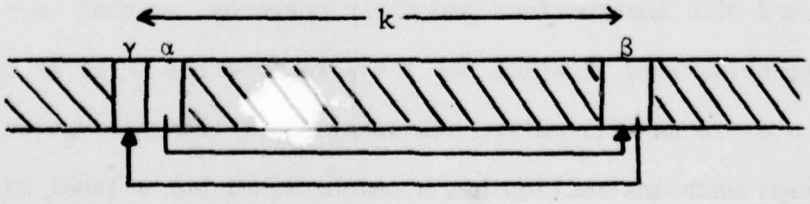


Figure 1

II. In-place Code Optimization

The simplest type of optimization problem on short/long address machine is to choose exactly those instructions which must have long addresses in a fixed piece of source code. This optimization could be performed, say, by an assembler for such a machine. It is clear that such an optimum uniquely exists: if the arrangement of instructions and data is fixed, making an instruction unnecessarily long cannot serve to shorten any others, although it may in fact force others to be long. Indeed, Figure 2 shows a case in which two instructions are arranged so that both must be short or both must be long. This is why the initial distance is defined with all instructions short--the following algorithm will not expand such situations unnecessary, but detecting them in order to collapse them could be difficult.

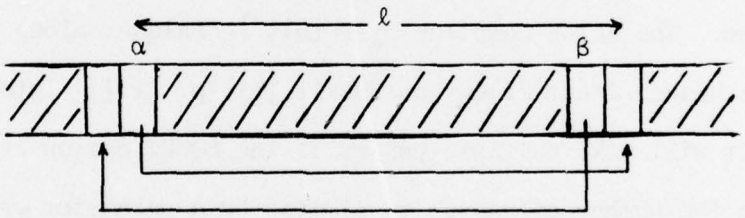


Figure 2: (Recall ℓ is maximum short address)

Theorem 1 A segment of fixed code may be optimized for short/long addresses in time proportional to $n \cdot \ell$, where n is the length of the code and ℓ is maximum short address.

A simple algorithm which almost suffices is to make repeated passes over the code, expanding on each pass only those instructions which require it, terminating after a pass in which no instructions are expanded. However, this may require about n/ℓ passes thru the program if instructions are chained as in the pathological case of Figure 3, where expanding δ forces

γ to expand which forces β to expand ...

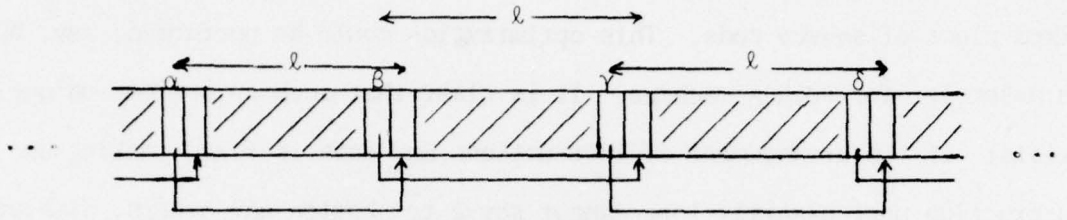


Figure 3

Such pathological cases require a better algorithm for theoretic reasons; but practicality also calls for a better algorithm not requiring many passes over the entire program, which is very likely stored out on a disk file. A number of systems, however, have made use of this multi-pass algorithm. The BLISS compiler uses this technique, along with others, to optimize branch instructions on PDP-11 [13, p. 119]. Interdata's CAL assembler will make multiple passes if the SQUEZ option is specified, although the number of passes is limited by a parameter on the option and assuming an initial long format thus missing cases as in Figure 2 [5, p. 53]. The UNIX assembler does three passes, also beginning with an assumed long format. Linear algebraic solutions, of high computational cost, are presented in [2 and 9]. Szymanski [12] gives an algorithm which is similar in flavor but n^2 in time.

The pathology of Figure 3 could, of course, be handled by a single right-to-left pass. However, the pathology can be elaborated to confound an alternating-pass scheme as well. The technique requires a single pass over the program to gather dependency information (called "initialization" below) and a single traversal through a generated data structure in a method reminiscent of topological sort [8].

ALGORITHM 1

```
data structures:
for each label α,
   a count C[α] and a list of labels L[α];
a queue Q;
/*initialization*/
for each label α, such that α:β,
   {set C[γ] ← d(α:β);
   for each γ:δ such that d(γ,δ) ≤ ℓ which is
      dependent on α, place γ on L[α];
   if C[α] > ℓ then place α on Q
   };
/*main loop - marking instructions as long or short*/
for each α on Q
   {output α;
   for each γ on L[α]
      {C[γ] ← C[γ]+1;
      if C[γ] = ℓ+1 then place γ on Q
   } }.
```

The output from this algorithm is the set of labels of instructions which must be in long format. The desired time bound is obtained on the main loop by observing that there are at most n labels and that $L[\alpha]$ has at most 2ℓ elements because of the test $d(\alpha,\delta) \leq \ell$ in the initialization loop. The initialization may also be implemented in a way which is linear in $n \cdot \ell$, by only searching for γ within ℓ of α on a single pass.

The above algorithm (or sketch) has the disadvantage of producing a potential enormous data-structure, with order of $n \cdot \ell$ entries. However, this sketch may be fleshed-out into a fairly efficient program. Rather than making initialization and marking distinct phases, they are combined in one pass. This pass is really an elaboration of the initialization, with marking done "on the fly". Fairly simple heuristics - such as checking whether an instruction is unquestionably long or short - are used to greatly reduce the number of labels for which lists must be created and the number of entries on these lists. Moreover, elements are removed from these lists at the first

opportunity, reducing the intermediate storage required. A complete algorithm incorporating these heuristics is given in the appendix.

An algorithm based on similar ideas is given by Szymanski [12]. He also notes that the algorithm will not work if operands of the form

label ± constant

are allowed. This is because lengthing one instruction may cause another to be closer to rather than farther from its operand. This happens in the pathological case where the addressed operand is on the other side of an instruction from the label specified in the operand - as in Figure 4 where lengthening γ causes $d(\alpha,\beta)$ to increase but moves the operand at $\beta + v$ closer to α. Indeed, Szymanski shows that problem of optimizing code with such pathologies is *NP*-complete. To call a practice such as this "bad coding" is certainly a severe understatement and any assembler should flag such usage as an error. This does point-out, however, the failure in this case of the common assembler convention that the difference of two relocatable terms is an absolute term.

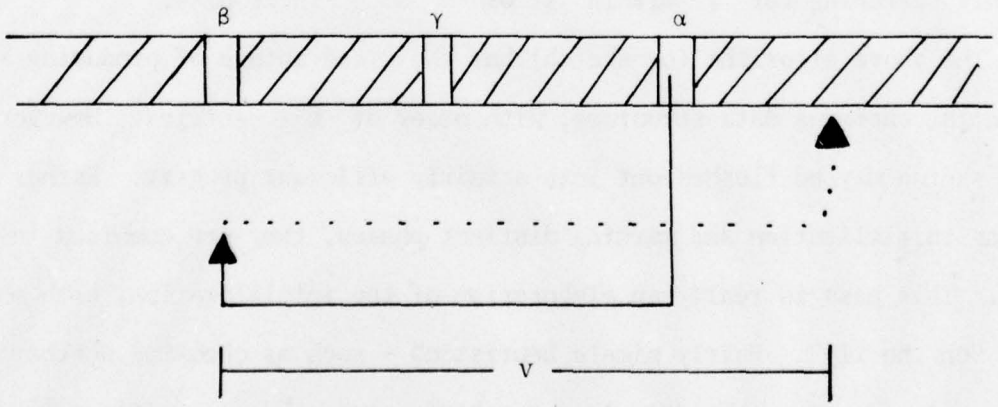


Figure 4

III. Optimization with Storage Allocation is Hard

Knowing that it is easy to optimize a program if no rearrangement of storage is possible, it is natural to ask whether rearranging storage may produce a shorter program and how difficult it is to perform such rearrangements optimally. The answer is, of course, that rearrangements may shorten a program but that finding an optimal arrangement is difficult. There are situations where this optimization is inappropriate or undesirable. Placing data and code together violates the tenents of "pure" code, making reentrant/recursive programming difficult to impossible. There are, however, occasions where it is highly appropriate, as in a compiler for a systems programming language for a single-user minicomputer system [4,10].

The situation we will consider is as above except that there are certain (one-word) variables whose location in the program body is not fixed *a priori* and certain places in memory (for example, following an unconditional branch) where these variables may be inserted. If a variable is referenced in only one brief segment of code, placing it near that segment could result in all references to it being short; placing it, say, at the end of the program would require all references to be long. In our graphical notation, a reference to such a variable will be indicated by an arrow pointing directly at a label (rather than a cell) and a slot where variables may be inserted will be indicated by a heavy vertical bar (see Fig. 5). Also, a number m in a cell will denote m adjacent occurrences of instructions with the same operand.

The problem of finding the optimal placement of variables in slots is shown to be hard is a sense which has already become classic in its brief lifetime: the problem is polynomially complete in *NP* or, more briefly, *NP*-complete. The *NP*-complete problems include many from combinatorics,

operations research, etc. whose best-known algorithms require exponential time. They are all related in that discovery a polynomial-time algorithm for any of the problems would provide a polynomial-time solution to all of them, and hence such an efficient algorithm is highly unlikely [1, 7].

In order to express this notion more exactly we define a problem $\langle S,P\rangle$ to be a set S (of instances of the problem) and a predicate P defined on the members of S. The set of the code generation problem (designated S/L-CODE) has instances of the form $\langle prgm,\ell,k\rangle$, where prgm is a source program and ℓ and k are integers. The predicate for code-generation (designated FITS) is then true if prgm may be translated into an object program with short address span ℓ which requires k or fewer instructions in long format. An optimal object program then requires the minimum k such that the predicate is true of $\langle prgm,\ell,k\rangle$. A problem $\langle S,P\rangle$ is said to be *NP*-complete provided that 1) for $s \in S$, it may be nondeterministically (by guessing) discovered that $P(s)$ is true in time polynomial in the size of s, and 2) for some fixed problem $\langle L,T\rangle$, from a known class of problems, there is a translation $tr:L\rightarrow S$ which operates deterministically in polynomial time, such that, for each $x \in L$, $P(tr(x))$ iff $T(x)$. The nondeterministic solution to the code generation problem, for an instance $\langle prgm,\ell,k\rangle$, first guesses the placement of variables, and then produces object code according to Algorithm 1, checking whether the resulting object code does indeed fit in k words. Thus short/long code generation satisfies the first criterion.

In order to show the second criterion, we need a fixed problem for reference, and we choose $\langle CNF\text{-}3,SAT\rangle$, where CNF is the set of all formulas of the propositional calculus which are in Conjunctive Normal Form with at most 3 literals per clause. That is a formula in CNF-3 is of the

form $C_1 \& C_2 \& \ldots \& C_m$, where each C_i is a clause of the form $A_1 \vee A_2 \vee A_3$ (or A_1 or $A_1 \vee A_2$), where A_i is a literal, that is either $\neg V$ or V for some variable V. $F \in$ CNF-3 is satisfiable (SAT(F) is true) if there is an assignment of truth values to the variables $V_1, \ldots, V_k$ of F such that F is true. This problem is known to be *NP*-complete [7].

We take advantage of the fact that the conjuncts $(V \vee \neg W)$ and $(\neg V \vee W)$ appearing in F force V and W to have the same value in any assignment of values satisfying F. Thus if a variable appears in more than five conjuncts, we may replace three occurrences of V by some new variable W and introduce the new conjuncts $(V \vee \neg W)$ and $(\neg V \vee W)$. This process is iterated until no variable appears in more than five conjuncts. The resulting formula is satisfiable iff the original one was. It is no more than three times the length of the original. Thus, without loss of generality, we assume that each formula of CNF-3 has no variable appearing in more than five conjuncts. Also we may easily assume that both a variable and its negation do not appear in the same conjunct.

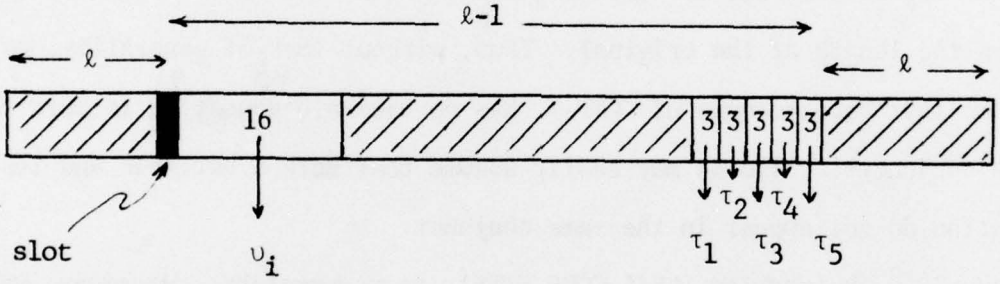
Theorem 2. The problem ⟨S/L-CODE,FITS⟩ is *NP*-complete. Moreover, we may restrict the maximum short address length ℓ to a fixed size and the problem is still *NP*-complete.

Proof: From the remarks above it suffices to show a translation: CNF-3 → S/L-CODE such that $F \in$ CNF-3 is satisfiable iff the corresponding program fits in the given space.

Pick an arbitrary formula $F \in$ CNF-3, say that F is $C_1 \& C_2 \& \ldots \& C_m$, where each conjunct C_j is $A_{j,1} \vee A_{j,2} \vee A_{j,3}$ (or $A_{j,1}$ or $A_{j,1} \vee A_{j,2}$, which are treated similarly and thus will not be distinguished). The variables of F are $V_1, V_2, \ldots, V_n$, and for each $V_i, c_{i,1}, c_{i,2}, \ldots, c_{i,5}$

(possibly fewer) are the conjuncts in which V_i appears. We first construct the program, which actually will be built-up of independent "modules"--two modules for each V_i and one for each C_j. The position of these modules does not influence the size of the object code in any way.

Figures 5 shows one of the modules (the f-module) corresponding to variable V_i. The other module (the t-module) is exactly the same except the occurrences of $\alpha_{i,j}$ and $\bar{\alpha}_{i,j}$ are reversed. Note that the distances shown are initial distances and that ℓ-1 is the initial distance from the rightmost of the three instructions pointing to τ_5 to the slot on the left.



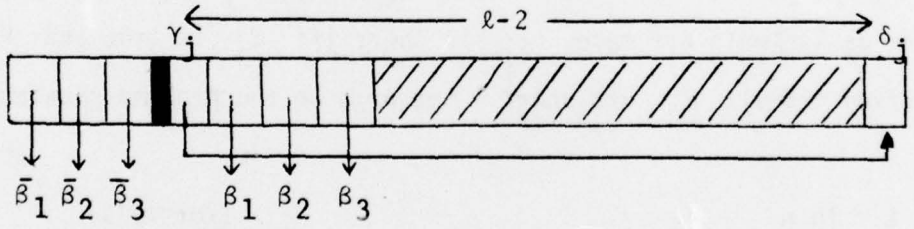
where τ_r is $\alpha_{i,c_{i,r}}$ if V_i is a literal of $C_{c_{i,r}}$
$\bar{\alpha}_{i,c_{i,r}}$ if $\neg V_i$ is a literal of $C_{c_{i,r}}$

Figure 5

Since υ_i is referenced only by the f- and t-modules of V_i and since it may be placed in the slot of either module without additional cost to that module, υ_i must be placed in one of these modules in any optimal

arrangement. We shall see that the placement of v_i in V_i's t-module corresponds to assigning the truth value <u>true</u> to V_i, and similarly the f-module corresponds to <u>false</u>. If v_i is placed in, say, the f-module then the variables labelled $\alpha_{i,j}$ or $\bar{\alpha}_{i,j}$ may be placed in the same slot so that all references to these variables from the module are short. In any optimal arrangement, the variables labelled $\alpha_{i,j}$ or $\bar{\alpha}_{i,j}$ must be placed in the f-module if such is the case, as will be shown later. On the other hand, if v_i is again placed in the f-module, then all instructions referencing it from the t-module must use long addresses. Thus 16 additional words are introduced between the slot of the t-module and the instructions referencing $\alpha_{i,j}$ or $\bar{\alpha}_{i,j}$, forcing all the corresponding addresses to be long in any case. Since the two modules corresponding to V_i are symmetric, the final combined length of these modules in an optimal arrangement is independent of the placement of v_i.

Now we turn to the module (c-module) corresponding to conjunct C_j which is shown in Figure 6, with C_j equal to $A_1 \vee A_2 \vee A_3$. First



where β_r is $\alpha_{i,j}$ if A_r is V_i
β_r is $\bar{\alpha}_{i,j}$ if A_r is $\neg V_i$
and $\bar{\beta}_r$ is similar with bars reversed

Figure 6

observe that any $\alpha_{i,j}$ or $\bar{\alpha}_{i,j}$ is referenced once in a c-module but three times in a t- or f-module. Moreover, omitting $\alpha_{i,j}$ from a c-module can cause at most one additional instruction, γ_j, to expand. Thus, if possible, the variables labelled $\alpha_{i,j}$ and $\bar{\alpha}_{i,j}$ must be placed in a t- or f-module for the arrangement to be optimal. We have seen that this only occurs if the variable ν_i is placed in the f- or t-module, respectively. Since $\alpha_{i,j}$ appears in the f-module of V_i iff $\bar{\alpha}_{i,j}$ appears in the t-module (and V_i is in C_j), exactly one of the variables $\alpha_{i,j}$ or $\bar{\alpha}_{i,j}$ is available for placement in the c-module of C_j. Thus, exactly one of the corresponding β_r or $\bar{\beta}_r$ is long and the final length of the c-module in an optimal arrangmenet is fixed except for the cell γ_j.

Again, consider the case that ν_i is placed in the t-module of V_i, and say V_i is a literal of C_j. Then, as noted above, $\alpha_{i,j}$ is available for placement in the c-module of C_j. Thus $d(\gamma_j,\delta_j) \leq \ell$ and hence γ_j may be short. If, however, none of the variables corresponding to β_1, β_2, or β_3 for C_j are available for placement in the c-module, the $d(\gamma_j,\delta_j) = \ell+1$ and γ_j must be long. As noted above, placement of ν_i in V_i's t-module corresponds to choosing true for V_i; and after all these assignments are made, δ_j is short iff C_j is true and F is satisfied iff all δ_j are short. Let prgm be the program constructed and let

$$k = 16 \cdot n \qquad \text{(for } \nu\text{'s)}$$

$$+ 4 \cdot \sum_{j=1}^{m} (\text{number of literals in } C_j) \qquad \text{(for } \alpha\text{'s and } \beta\text{'s).}$$

Then SAT(F) iff FITS(prgm,ℓ,k).

Inspection of Figures 5 and 6 will show that an ℓ of $31 (= 2^5-1)$ is sufficient. □

One may suspect that the constraint that variables can only be placed in specified slots makes the problem difficult. However, this is not the case. We define two variations on the rules governing the arrangement of program. FITSa (FITS with arbitrary placement) is true of <prgm, ℓ, k> if the object program may be arranged as before except that variables may be inserted anywhere in the program text, and the resulting program requires k or fewer words in long format. Realistically we should charge an extra word for branching around the inserted variable, but it is not necessary in the following result. It will be necessary to charge for branching in FITSb (FITS with branching) where the code may be rearranged to bring it near the variables which are referenced. Figure 7 illustrates a situation where this may be an advantage. The blocks labelled

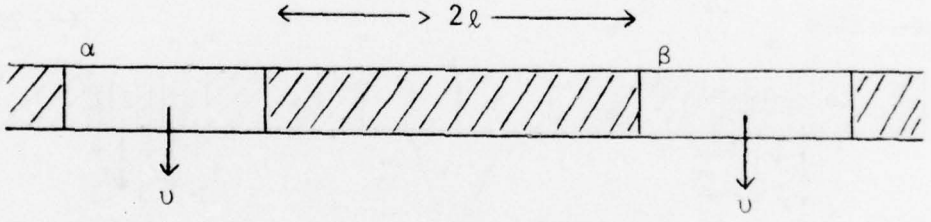


Figure 7

α and β both reference the variable υ, but they are too far apart for υ to be referenced by short instructions for both. However, these blocks could be rearranged as in Figure 8, with br indicating branch instructions. FITSb (<prgm, ℓ, k>) is true if prgm fits in k words of object code with rearrangements as indicated, provided that the cost of branch instruction is also accounted for.

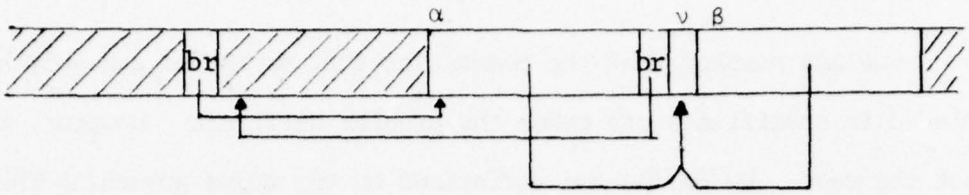
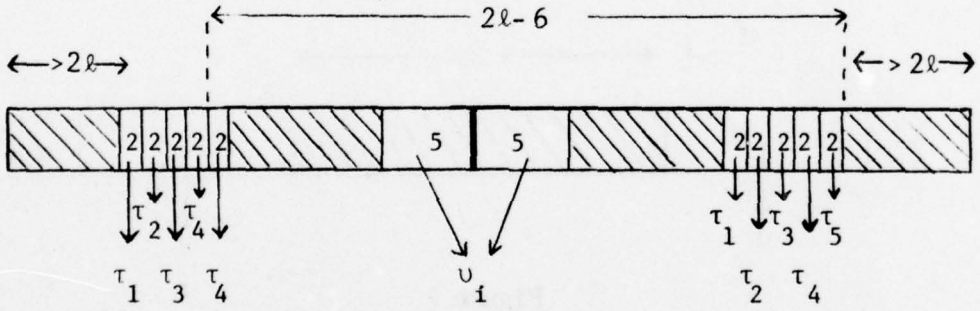


Figure 8

Theorem 3 The problem <S/L-CODE,FITSa> is *NP*-complete.

Proof: The proof is similar to that of Theorem 2, except that the modules corresponding to the V_i (Fig. 5) must be modified. This modification is essentially accomplished by mirroring the module about the slot, as shown in Figure 9. In addition, each reference to β_r or $\bar{\beta}_r$ in a c-module (Fig. 6) is doubled.



where τ_j is $\alpha_{i,j}$ if V_i is a literal of C_{c_j}.

$\bar{\alpha}_{i,j}$ if $\neg V_i$ is a literal of C_{c_j}.

Figure 9

As before, placing υ_i in the t-module of V_i corresponds to assigning true to V_i, etc. If υ_i is placed in the center of the module, then each τ_j may also be placed in the center of the module in such a way that it may be addressed with short instructions from

either end, thus "saving" four words. If the τ_j is placed in its corresponding c-module, however, at most three words can be saved, thus forcing the placement in the t- or f-module if possible.

If, on the other hand, υ_i is not placed in, say, a t-module, then 2ℓ words will separate the instructions referencing τ_j from the left and right ends of the module. Thus placing τ_j in the t-module will save only two words, while at least two words will be saved by placing τ_j in the corresponding c-module. Thus, although the placement of τ_j is not forced in this case, its placement in the c-module is no worse. The rest of the analysis follows as in Theorem 2. □

Theorem 4 The problem <S/L-CODE,FITSb> is polynomially complete in *NP*.

Proof: The proof is essentially that of Theorem 3, except that two instructions referencing addresses local to the module are inserted between every reference to a variable υ_i, $\alpha_{i,j}$, etc. This forces the entire module to remain together, since moving a segment containing intra-module references as well as references to variables costs two words in intra-module addresses for every word saved addressing a variable, and since only moving references to variables costs two branch instructions (at least two words) for every reference.

Actually some care must be taken in constructing the intra-module references. For example, if all these references pointed to just one address, that address could be picked up and moved as well. However, judicious construction, chaining intra-module references, can solve this problem. It is sufficiently a bookkeeping detail that it will not be further elaborated on. □

Although negative results from computer science theory are usually received with distain or dismay, the last two theorems should be cause for some rejoicing. Most computer programmers have already had sufficient headaches dealing with "optimizing"* compilers that they would be most dismayed at an algorithm giving compilers license to rearrange their code (even with branch statements inserted).

---

*The author's edition of Webster's New World Dictionary (World Pub., 1957) gives "to be given to [or] to treat with optimism" as the only definition for "optimize."

IV. APPROXIMATE SOLUTIONS

When we discover that a particular problem class is *NP*-complete, or in some other way difficult to solve, it is natural to ask whether there is some algorithm which provides an approximate solution [3,11]. Certain problems, such as satisfiability of a Boolean formula, have only "true" or "false" as answers and hence do not admit approximate solutions. However, if a problem is, say, one of minimization it is reasonable to ask how close can we come to the minimum with an algorithm which is "inexpensive" to run. In such a case, we are interested in the relative rather than the absolute "error". In particular, a problem is said to have an ε-approximate algorithm ($\varepsilon > 0$) if, for each instance of the problem

$$\left| \frac{\$_A - \$_O}{\$_O} \right| < \varepsilon$$

where $\$_A$ is the cost of the solution provided by the algorithm and $\$_O$ is the true optimal solution. Since we will concern ourselves only with minimization of positive quantities, the above inequality is equivalent to $\$_A \leq \$_O(1+\varepsilon)$.

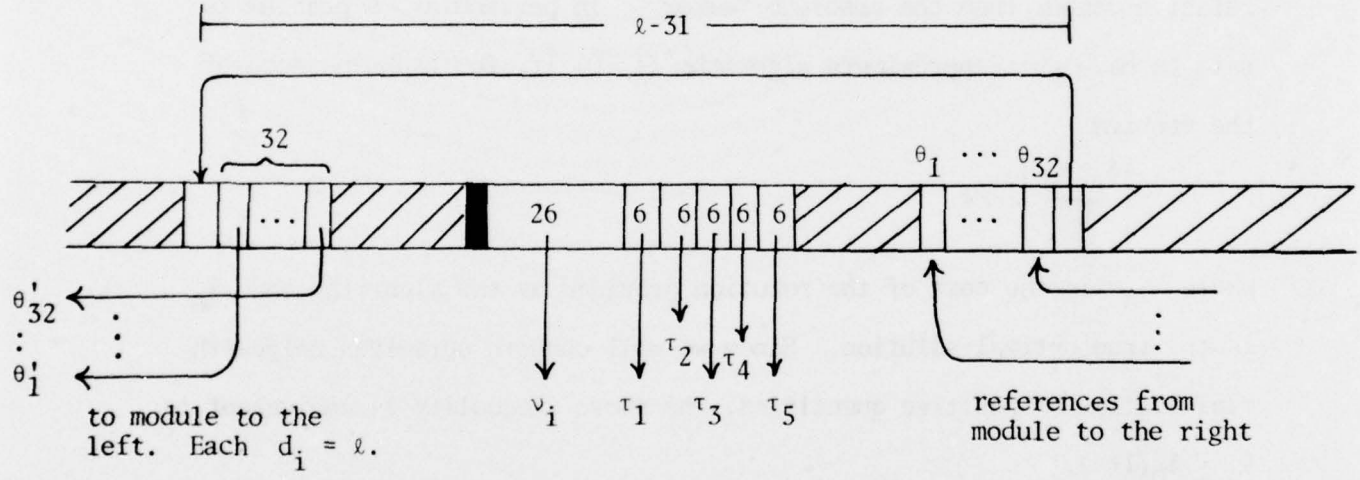
Theorem 4. For any $\varepsilon > 0$ the problem of finding ε-approximate solutions to ⟨S/L-CODE,FITS⟩ is *NP*-complete. Moreover, ℓ may be fixed.

Proof: Once again the proof is by reduction to CNF-3 with the same restrictions as before.

Say $\varepsilon > 0$ is fixed and we are given a formula F as above.

As before, we constrict a t-module and an f-module (Figure 10) for each variable V_i and a c-module for each clause C_j (Figure 11).

Whereas these modules had been independent in previous proofs, they will now be linked together such that certain instructions in each module reference operands in the module immediately to the left. The order in which the modules are arranged is of no consequence, but it is important that they are linked properly. In particular, the initial instruction-operand distance of each inter-module instruction is exactly ℓ - this condition is achieved by having a sufficient number of labels (maximum 32) in each module and by providing sufficient "filler" between modules. The extra-module instructions of the leftmost module are replaced by no-ops and a special module will be placed on the right of the entire assemblage.



where τ_r is $\bar{\alpha}_{i,c_{i,r}}$ if V_i is a literal of $C_{c_{i,r}}$

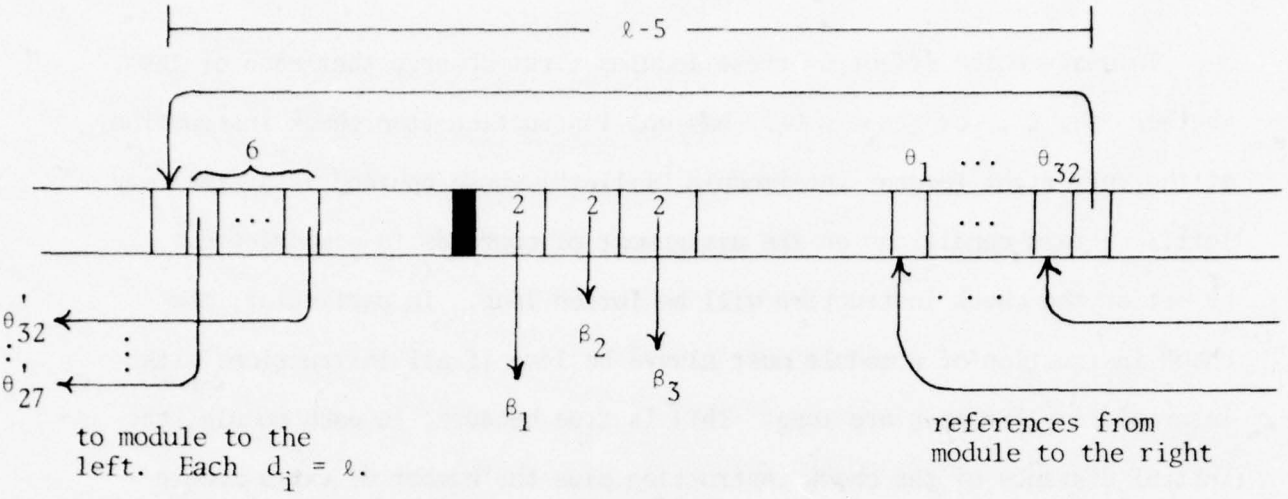
$\alpha_{i,c_{i,r}}$ if $\neg V_i$ is a literal of $C_{c_{i,r}}$

the θ's are unique to each module

Figure 10

To analyze the effect of these modules first observe that each of them, whether t-, f-, or c-module, has one instruction (the check instruction) at the very right (except intermodule "filler") whose operand is at the very left. Certain conditions on the assignment of operands to a module must be met or the check instruction will be forced long. In particular, the check instruction of a module must always be long if all instructions with intermodule references are long. This is true because, in each module, the initial distance of the check instruction plus the number of extra-module references in $\ell + 1$, and because all extra-module references come from within the span of the check instruction. Moreover, since the check instruction is with the span of all references from the module to the right, and since each of these references from the right has initial distance ℓ, making the check instruction of one module long forces the check instruction of the right neighbor to be long as well. In this way lengthening one check instruction propagates to lengthening the rightmost check instruction.

Consider now the f-module for V_i, and assume the check instruction of this module has not already been forced long by a module to the left. This check instruction may remain short if either the operand v_i or the operands corresponding to each of the τ_r's are assigned to the module. Since both the t-module and the f-module of V_i reference v_i, and since the intermodule filler of length ℓ prevents v_i from being near both modules, one of the modules must be assigned v_i while the other its corresponding τ_r's. Say v_i is assigned to the f-module, then $\bar{\alpha}_{i,j}$ is available for the c-module of any C_j in which V_i appears as a literal. And the c-module for C_j must have assigned to it at least one operand corresponding to a β_r, or its check instruction will be forced long.



where β_r is $\alpha_{i,j}$ if A_r is V_i

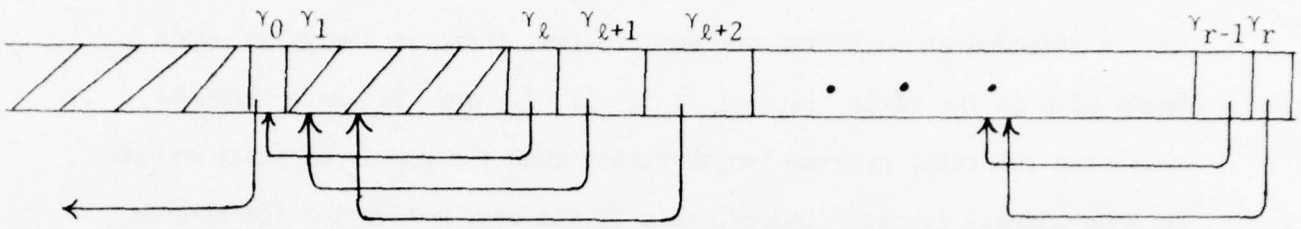
$\bar{\alpha}_{i,j}$ if A_r is $\neg V_i$

the θ's are unique to each module

Figure 11

To sum up the construction so far, the assignment of v_i to the corresponding t- on f-module corresponds to an assignment of <u>true</u> or <u>false</u>, respectively, to V_i, and the truth assignment satisfies the formula iff no check instruction is forced long. If the formula is unsatisfiable, then the rightmost check instruction must be long.

Finally we a block of q instructions(the exact value of q is given later) labelled $\gamma_1,\gamma_2,\ldots,\gamma_q$ (Figure 12). If θ_{31} is the indicated label from the rightmost module, the $\gamma_1 : \theta_{31}$ and enough filler is provided so that $di(\gamma_1,\theta_{31}) = \ell$. The $\ell - 1$ instructions $\gamma_2,\ldots,\gamma_\ell$ have no operands and $\gamma_r : \gamma_{r-\ell}$ for $\ell < r \leq q$. The placement and construction of this block is such that γ_1 is forced long iff the rightmost check instruction



each initial distance exactly ℓ

Figure 12

is forced long; and forcing γ_1 long forces each of $\gamma_{\ell+1},\ldots,\gamma_q$ to be long in turn. Thus each of these $q - \ell$ γ's must be long iff F is not satisfiable. Moreover, either all of these γ's must be long or none must be.

Now say the constructed program, before adding the γ's, has p short/long instructions. Then let

$$q > p(1 + \varepsilon) + \ell .$$

Now assume F is satisfiable. Then an optimal solution has cost $\$_0$ which is less than or equal p, as does any solution which does not force the γ's to be long. However, if the approximate solution forces the γ's to be long, then

$$\$_A \geq q - \ell > p(1 + \varepsilon) \geq \$_0(1 + \varepsilon) .$$

Hence any ε-approximate algorithm must provide a solution of cost less than an equal to p if the orginal F was satisfiable. And this indeed would provide a reduction solving 3-CNF.

Examining Figure 10, we see that the f- and t-modules, which are larger than the c-modules, may have an initial length as short as 120, and hence an ℓ of 151 suffices. □

Because of the fact that an instruction in a short/long address machine can be expanded at most from one word to two, there is always an upper bound of 1 on the ratio $|\$_A - \$_O/\$_O|$, if $\$_A$ and $\$_O$ were taken as measuring the total program length rather than the number of instructions in long-address format. Indeed, this is the very motivation for counting long instructions rather than total program length. Indeed, if we modify the predicate FITS so that FITSt(prgm,ℓ,k) is true if prgm may be assembled in k or fewer words (total program length) on a machine with short address span of ℓ, then we have:

Theorem 5 For any ε, $0 \leq \varepsilon < 1$, the problem of finding -approximate solutions to ⟨S/L-CODE,FITSt⟩, with fixed ℓ, is *NP*-complete. However, an ε-approximate solution for $\varepsilon = 1$ trivially exists.

Proof: Perform the construction exactly as in Theorem 4, except that p is the total length of t-, f-, and c-modules, filler, and associated variables - that is everything except the block of γ's. Now add a block of q γ's where

$$q > (\ell + (1 + 2\varepsilon)p)/(1 - \varepsilon) .$$

Since $0 \leq \varepsilon < 1$, $1 - \varepsilon$ is always positive so this restriction is equivalent to

$$q > \ell + p + \varepsilon(2p + q) .$$

Now assume F is satisfiable. Then

$$\$_O \leq 2p + q .$$

However, if an approximate solution requires the γ's to be in long format, then

$$\begin{aligned} \$_A &\geq (q - \ell) + p + q \\ &> p + \varepsilon(2p + q) + p + q \geq (1 + \varepsilon)\$_O . \end{aligned}$$

Hence any ε-approximate solution for a program corresponding to a satisfiable F must have the γ's in short format. Again, this would provide a reduction for 3-CNF.

As noted above, $\$_A \leq 2 \cdot \$_O$ from the nature of the problem. The of Theorem 4, $\ell = 151$, again suffices. □

V. Heuristic Considerations

The previous section demonstrated that even approximation of a rather broadly constrained version of the problem of code generation was *NP*-complete. The implication of such complexity results is not a purely negative one, for it directs our research to heuristics and approximations which are likely to, but are not guaranteed to, provide near-optimal code. In this section we begin this attempt, with heuristics for the placement of varibles during compilation or assembly, and considerations and suggestions for future architecture, language, and program designers.

Heuristics for the placement of variables can be exercised by the programmer or can guide manipulations by an assembler or compiler. The prime consideration for a programmer is to keep logically local variables allocated locally. A common-place technique, which is certainly economical in FORTRAN on a large machine, is to declare a few temporary variables which are used repeatedly throughout a program--using few words instead of many. However, in machines with instructions of short/long format variation this is potentially quite wasteful. Local variables should be used as much as possible, with small local blocks if the language permits.

Corollary to the programmer heuristic for local variables is one recommending placement of loop indices within/adjacent-to loops. Unless testing is performed in two different places, a for-loop structure must have an unconditional branch, and the loop index may be placed immediately following this. This may be done by the programmer in assembly language or by the compiler in a language like ALGOL W (except that ALGOL W is recursive) in which a loop-index is implicitly declared for the body of the loop.

Automatic heuristics, applied by an assembler or compiler, presuppose the freedom to rearrange storage. The natural "greedy" heuristic is to

count the number of times each variable is referenced by each module and to associate the variable with the module which references it most frequently. Considerations of branch instruction are omitted. Not only are "far away" branches relatively unlikely in a well-written program, but their impact on program execution time (related to instruction formats because long instructions require additional time or complexity of instruction fetches) is even further restricted. This is because frequently executed branches are in tight-loop situations and hence are nearby. Exceptions to the above are subroutine branches, but subroutines in this context should perhaps be treated as data objects whose placement is optimal. Moreover, the "greedy" heuristic ignores the fact that the placement of variables will be significant, in that one variable placed between an instruction and its operand could force that instruction to be long (as in sections III and IV). In general this is not worth considering for simple variables; but arrays and other structured variables should be weighted by the number of references divided by size, and objects of highest weight placed with the module first. This is intentionally a vague specification, since addressing modes, the definition of "module", language features, etc. will strongly influence the actual details. It is easy to imagine instances where this heuristic is arbitrarily bad (as is expected from section IV). However, the heuristic can be implemented without much additional overhead as part of a first pass.

If we wish to consider branches and the interactions of instructions, we may estimate a cost for each short/long instruction $\alpha:\beta$ which is a function of all $\gamma:\delta$ dependent on $\alpha:\beta$ considering a) the cost of $\gamma:\delta$, b) $\ell - d(\gamma:\delta)$, c) a stage in an iterative approximation of the cost (since dependency is very likely to be cyclic). A similar cost for each possible

placement of a data object could be computed and then a modified greedy heuristic could maximize estimated benefit/cost. This cost estimation could require time proportional to n^2, however.

Any automatic heuristics for placement of data require language features which permit movement of data objects. One might assume that an assembler could always place variables after unconditional branches, but this could wreak havoc in the middle of a branch table (or in other instances of calculated addresses, albeit not ones appealing to current aesthetics). Many assembler languages have "literal pool here" pseudo-operations, and a similar pseudo-operation could allow the programmer to declare sites in the code where storage for variables could be reserved. Scoping might present some problems, but there is no reason an assembly language cannot be block structured. Or a notation for label temporaries (comparable to the "3F" and "7B" labels in MIX [8]) could distinguish names for which the assembler was to allocate storage from those the programmer wished to control. Higher-level languages of course free the allocation strategy from concern about explicitly generated addresses, but for those which dynamically allocate storage or force program and data separation these considerations are of course irrelevant. However, a FORTRAN compiler could perform optimization not only with user variables but with its own temporaries which it may use during expressions evaluation, etc.

Consideration of block structured but non-recursive languages points-up what may be as much a deficiency in machine architecture as in language. With block-structure the programmer has considerable ability to restrict scope of temporaries, but the common $\pm\ell$ addressing range restricts the useful size of blocks with local variables. The usual block structure

restricts declarations to beginning of blocks, and it is natural for compilers to allocate storage immediately (and necessary for a one-pass compiler). This means that data references may never make use of the forward range of short instructions. Three solutions to this seem to exist. One lies with the compiler, at the expense of considerable complication and perhaps another pass. This is to allocate storage in the middle of the block. The second solution is architectural and not out-of-the question on microprogrammed machines. This is to have short data references in the range -2ℓ to 0, perhaps retaining the $-\ell$ to $+\ell$ range for branches. Finally, short data references may be displacement from a base address in a system or programmer controlled register.

Another observation about short/long addresses is that indirection (as in the PDP8) is better than purely an extra word with a long address. Indirection within the $\pm\ell$ range to a word containing the full address of a variable allows many instructions to share that full address. If a page 0 or other common segment is addressable by a short instruction, even more sharing is possible. Indeed the Prime system makes use of such a shared with addresses placed at link/load time. Optimization at link/load time is an interesting area for general consideration.

Concern about execution (i.e., fetch) time rather than merely program size has been mentioned before the preceding paragraphs, but it is fitting that this section should end stressing this point. It would be quite foolish, in execution cost, for an allocation strategy to place a variable with a routine executed only on rare exceptions rather than with a major loop. Therefore, it is necessary either to predict probabilities of program flow based on reasonable programming style or to provide facilities for the

programmer to indicate allocation preferences (cf. the FREQUENCY statement of early FORTRAN). There may even be a time-space tradeoff, where constants or modules a replicated to shorten instructions and thus save execution time, but where the replication uses more space than is saved by short instruction formats.

REFERENCES

[1] S. A. Cook, "The complexity of theorem proving procedures", Proc. 3rd Annual ACM Symp. on Theory of Computing (May 1970), pp. 151-158.

[2] G. Frieder and H. J. Saal, "A process for the determination of addresses in variable length addressing", Comm. ACM 19, 6 (June 1976), 335-338.

[3] M. R. Garey and D. S. Johnson, "Performance guarantees for heuristic algorithms", Algorithms and Complexity: New Directions and Results (J. F. Traub, ed.), Acad. Press, New York (1976), 41-52.

[4] K. Harris, "The design and implementation of a complier for a mini-computer", M.S. paper, Computer Science Dept., Pennsylvania State University (Aug. 1976).

[5] Interdata Corp., Common Assembler Language User's Manual, Oceanport, N.J., 1974.

[6] D. S. Johnson, "Approximation algorithms for combinatorial problems", J. Comput. Syst. Sci. 9, 3 (Dec. 1974), 256-278.

[7] R. M. Karp, "Reducibility among combinatorial problems", in Complexity of Computer Computations (Miller and Thatcher, eds.), Plenum Press, N.Y., 1972.

[8] D. A. Knuth, The Art of Computer Programming, Vol. I: Fundamental Algorithms, Addison-Wesley, 1968.

[9] D. L. Richards, "How to keep addresses short", Comm. ACM 14, 5 (May 1971), 346-349.

[10] P. Roche, "Code generation for PL 1600", M.S. paper, Computer Science Dept., Pennsylvania State University (Feb., 1977).

[11] S. Sahni and T. Gonzalez, "P-complete approximation problems", J. ACM 23, 3 (July 1976), 555-565.

[12] T. G. Szymanski, "Assembling code for machines with span-dependent instructions", Tech. Rept. 224, Dept. of Elect. Engr., Princeton University, Dec. 1976.

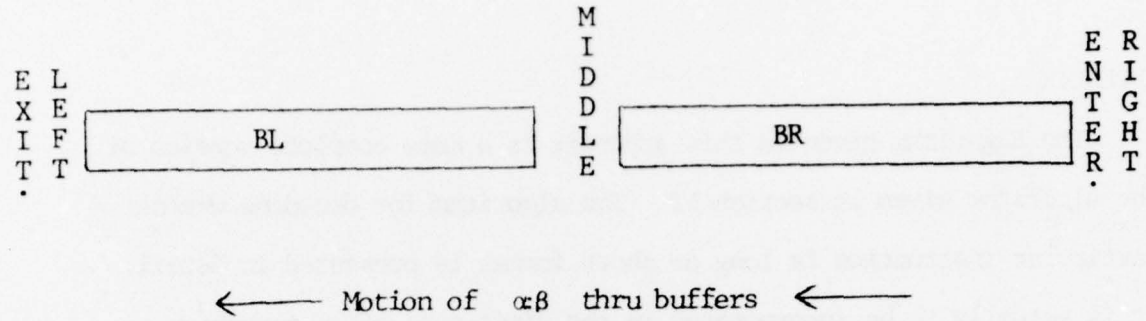
[13] W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, The Design of an Optimizing Compiler, Amer. Elsevier, New York, 1975.

APPENDIX

The algorithm given in this appendix is a more complete version of the algorithm given in section II. The algorithm for deciding whether a particular instruction is long or short format is presented in detail. It is actually to be incorporated in the first pass of an assembler, handling instructions after they have been processed by the scanner, symbols entered into the table, etc. For simplicity, the algorithm assumes that all assembly language instructions are of the same form -- single operand with the option of either short or long format machine representations.

The critical idea for the heuristics of the algorithm is to determine immediately, if possible, whether an instruction must be short or long format independant of the formats of all other instructions -- which we call "certainly" long or short. An instruction is "certainly long" if at least ℓ words of source program (generally assumed to be in short format) intervene between the instruction and its operand. On the other hand, an instruction is "certainly short" if the number of intervening words plus the number of intervening instructions whose format is yet unknown (and which, therefore, could expand requiring an extra word) is less than ℓ. It is, of course, easy to test these conditions for instructions which reference backwards, but a forward reference requires an ℓ word "look-ahead."

The "look-ahead" is actually accomplished by ℓ instruction buffers (see Figure 13). The buffers are ℓ instructions rather than merely ℓ words in length in order to facilitate synchronization of the processes filling and emptying the buffers. Backward reference instructions are classified as certainly long or short by ENTER.RIGHT, which accepts all input and fills the



ENTER·RIGHT	accepts inputs from scanner and fills BR
MIDDLE	empties BR and fills BL
EXIT·LEFT	empties BL and writes output

Figure 13: Buffers and procedures for Algorithm II

right buffer BR. The procedure MIDDLE takes input off BR and places instructions on BL, the left buffer, after processing. Forward reference instructions can be classified certainly long/short by MIDDLE, since if the operand is not among the ℓ instructions in BR then the format is certainly long. MIDDLE also places all backward reference instructions, except those whose format is known based on the information in BL, into the list structures corresponding to those set up during the initialization phase of Algorithm I (this is part of the initialization "on the fly"). Deferring the entering of these instructions until this point means that all certainly long/short instructions in their scope have been discovered. For a similar reason, forward reference instructions are not entered on to the lists until they are removed from BL by EXIT·LEFT.

Since the length of generated code appears to increase as more instructions are forced to expand to long format, each of the three procedures

ENTER·RIGHT, MIDDLE, and EXIT·LEFT keeps its own location counter (at least potentially, the location counter is not explicitly kept in EXIT·LEFT, but is kept in ENTER·RIGHT and MIDDLE as PR and PM respectively). By biasing these location counters by distinct multiples of 100000 (presumed to be longer than ℓ or the length of any generated code), it is possible to make comparisons which test the relative position of two labels even though they have been assigned positions by different procedures.

where label gets position value	bias
undefined	300000
ENTER·RIGHT	200000
MIDDLE	100000
EXIT·LEFT	0

Table 2

For example, the test

$$P[\alpha] < P[\beta]$$

is actually a combination of two tests: has α passed a "station" in the buffers which β has not, or have they both passed the same station with α preceding β. Of course it is occassionally necessary to ensure the same bias is used when comparing two positions -- hence the two different times when P and N are assigned values in the procedure MIDDLE, immediately in the case of a backward reference but not until the completion of processing a forward reference.

We now give the algorithm in a slightly unconventional but hopefully understandable notation. In particular, manipulations of lists, buffers, files, queues, and records (the preferred implementation of C,D,P,L[α]) are largely descriptive.

```
/* ==== data structures ==== */


buffer BR, BL for ∠ instructions;


for each label ∝ {
   integer C[∝] initially 0   /* count */;
   integer D[∝] initially 0   /* distance */;
   integer P[∝] initially 300000 /* position */;
   list    L[∝] initially null };


/* PR and PM are position estimates for right and middle,
respetively, biased by their initial values. Undefined
position, in P[•], is 300000 */

integer PR initially 200000;
integer PM initially 100000;


/* NR and NM count the number of yet unmarked instructions
(i.e. instructions for which the required length has not
been determined) at the right and middle respectively */

integer NR, NM initially 0;
```

```
      /* ==== specialed procedures ==== */

Boolean procedure LONG(θ,φ)
   /* returns true IFF θ:φ is certainly long
   from currently known information*/
   |P[θ] - P[φ]|  >  ℓ;


Boolean procedure SHORT(θ,φ)
   /* returns true IFF θ:φ is certainly short */
   |P[θ] - P[φ]| + |N[θ] - N[φ]|  ≤  ℓ;


procedure LONG•DELIST(*)
   /*corresponds to the main loop of algorithm I.  The
   instruction labelled ε is made long ("marked") and the
   bookkeeping of all γ:δ which have ε in their scope is
   updated.  If this forces γ:δ to be long, then the
   procedure is repeated with γ, etc.*/
   { queue Q;
   put * on Q;
   for each γ on Q
      {mark γ as long;
      for each η on L[γ]
         {D[η] <- D[η] + 1; C[η] <- C[η] - 1;
         if D[η] = ℓ + 1 then place η on Q}
   }  };


procedure SHORT•DELIST(*)
   /*similar to the main loop of algorithm I and the
   procedure LONG•DELIST, except the instruction labelled ε
   is made short */
   { queue Q;
   put * on Q;
   for each γ on Q
      {mark γ as short;
      for each η on L[γ]
         {C[η] <- C[η] - 1;
         if C[η] + D[η] = ℓ then put η on Q}
   };
```

```
      /* ==== main "coroutines" ==== */

procedure ENTER•RIGHT
   { α:β gets next instruction from file;
   P[α] <- PR; N[α] <- NR;
   PR <- PR + 1;
   if P[α] - P[β] > K  /*only backwards ref. sat'fy*/
      then  { mark α:β as long; PR <- PR + 1}
      else NR <- NR + 1;
   place α:β on BR};


procedure MIDDLE
   { PM <- PM + 1;
   α:β <- next instruction from BR; place α:β on BL;
   if α:β is marked long
      then PM <- PM + 1
   else if P[α] < P[β] /*forward reference */
      then {
      if LONG(α,β) /*compare based on PR values */
         then { PM <- PM + 1; mark α:β as long }
      else if SHORT(α,β)
         then mark α:β as short
      else
         NM <- NM +1;
      P[α] <- PM; N[α] <- NM}
   else  /* backward reference */
      { P[α] <- PM; N[α] <- NM;
      if LONG(α,β) /*based on new values*/
         then { PM <- PM + 1; mark α:β as long }
      else if SHORT(α,β)
         then mark α:β as short
      else
         {integer T initially 0;
         for each unmarked γ:δ on BL such that P[γ] > P[β]
            { T <- T + 1; enter α on L[γ]};
         C[α] <- T; D[α] <- P[α] - P[β];
         NM <- NM + 1}
   };
```

```
procedure EXIT•LEFT
   { α:β <- next instruction from BL;
   if α:β is unmarked and P[α] < P[β]
      then
      if LONG(α,β)
         then LONG•DELIST(α)
      else if SHORT(α,β)
         then SHORT•DELIST(α)
      else
         { integer TN, TP initially 0;
         for each γ:δ on BL such that P[γ] < P[β]
            if γ:δ is marked short
               then TP <- TP + 1
            else if γ:δ is marked long
               then TP <- TP + 2
            else
               { TP <- TP + 1; TN <- TN + 1};
         if TP + TN ≤ ℓ
            then SHORT•DELIST(α)
         else
            for each γ:δ on BL such that P[γ] < P[β]
               if γ:δ is unmarked
                  then enter α on L[γ];
         D[α] <- TP; C[α] <- TN }
   };
```

```
      /* ==== mainline program ==== */

/*  the main  program  is essentially  cycling throught  the
three  "coroutines"  ENTER•RIGHT,  MIDDLE,  and  EXIT•LEFT.
There  is, however,  some  initial and  final control  since
ENTER•RIGHT must be executed ℵ times before the right buffer
is filled and MIDDLE begins executing, etc. */

/* initial filling of buffers */
for N ∈- 1 to ℵ
   do ENTER•RIGHT;
for N ∈- 1 to ℵ
   do { ENTER•RIGHT; MIDDLE };

/* process further input */
while INPUT•REMAINING
   do { ENTER•RIGHT; MIDDLE; EXIT•LEFT };

/* final flushing of buffers */
for N ∈- 1 to ℵ
   do { MIDDLE; EXIT•LEFT };
for N ∈- 1 to ℵ
   do EXIT•LEFT.
```

The above algorithm, as specified, provides a method of deciding whether each instruction is short or long format, but does not apply this information to the actual generation of instructions. A particular problem is the fact that the information about instruction length, in either version of the algorithm, is not produced in the order of the program text. This would not be particularly important if the label α of instruction $\alpha:\beta$ was actually a user-defined label and hence resided in the symbol table. But α is likely to be "generated" by the assembler -- for example the statement number --and there would be far too many generated labels to fit in a reasonably sized symbol-table.

One of the major advantages of the above algorithm, however, is that the format of "most" instructions is determined before they leave the left buffer. A final tentative location counter (PL is the above notation although not actually part of the algorithm) would be kept by EXIT·LEFT and assigned to $P[\alpha]$ as $\alpha:\beta$ leaves the buffer if α is a user defined symbol. Still assuming unmarked instructions are short, it is however necessary to adjust for instructions discovered to be long by LONG·DELIST after they have left LB.

In particular, a file FIXUP should be created to hold adjustment information and the statement

"mark γ as long"

in LONG·DELIST should be augmented to

"if γ is in BL then mark γ as long
else output γ on file FIXUP".

At the end of the first assembly pass the file FIXUP is then sorted in statement number. A linear pass thru this sorted file will then determine an adjustment factor which is added to the location of user-defined symbol.

This adjustment can be thought of as the operation

> "if the statement number of user defined label γ is in the range S_i to S_{i+1} then the actual position of γ is $P[\gamma] + a_i$".

It is, of course, efficiently implemented as a binary search. "Origin" psuedo operations which reset the assemblers location counter must also be included in FIXUP if they are more complicated than setting the location counter to a constant or to the old value of the location counter plus/minus a consort. Few uses of "origin" pseudo-operations would violate these conditions. They must in any case be restricted to expressions containing user defined symbol(s) which have been previously defined in the program text, for otherwise extra pass(es) would be required. The user defined variables will thus have adjustment already associated with them, which can be used to compute the adjustment corresponding to the "origin."

(14) MRC-TSR-1779

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 1779 (9) Technical summary rept.	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): CODE GENERATION FOR SHORT/LONG ADDRESS MACHINES,	5. TYPE OF REPORT & PERIOD COVERED: Summary Report - no specific reporting period
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): (10) Edward L. Robertson	8. CONTRACT OR GRANT NUMBER(s): (15) DAAG29-75-C-0024, NSF - MCS-76-17323
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin, 610 Walnut Street, Madison, Wisconsin 53706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: # 8 (Computer Science)
11. CONTROLLING OFFICE NAME AND ADDRESS: See Item 18 below.	12. REPORT DATE: (11) Aug[illegible] 1977
	13. NUMBER OF PAGES: 42 (12) 46p.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

National Science Foundation
Washington, D. C. 20550

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

short/long addresses
span-dependent instructions
variable-length addressing
assembler
compiler
code-generation
optimization
computational complexity
NP-completeness

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Many machine languages have different instruction formats which allow addressing of "nearby" operands with a "short" instruction (e.g. one word), while "faraway" operands require a "long" format (e.g. two words). Because the size of object code may depend upon the formats used, the formats of different instructions may be dependent on each other.

An efficient algorithm is given for optimally assigning formats to instructions in a given program, and an implementation will be discussed which is practical in space as well as time. The more sophisticated problem of arranging

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

221200

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT - Cont'd.

operands within programs is discussed. Unfortunately, it is unlikely that an efficient algorithm can even guarantee good approximations for this problem. Finally, implications of this problem on hardware and software designs are considered.