Code Generation and Storage Allocation for Machines With Span-Dependent Instructions

EDWARD L. ROBERTSON
Indiana University

Many machine languages have two instruction formats, one of which allows addressing of "nearby" operands with "short" (e.g. one word) instructions, while "faraway" operands require "long" format (e.g. two words). Because the distance between an instruction and its operand depends upon the formats of the intervening instructions, the formats of different instructions may be interdependent.

An efficient technique is discussed which optimally assigns formats to instructions in a given program and 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 guarantee even a good approximation for this problem, since it is shown that r-approximation is NP-complete.

Finally, implications of these problems for hardware and software design are considered.

Key Words and Phrases: short/long addresses, span-dependent instructions, variable-length addressing, assembler, compiler, code-generation, storage-allocation, optimization, computational complexity, NP-completeness
CR Categories: 4.11, 4.12, 5.25

1. INTRODUCTION

One aspect of machine organization which occurs quite frequently in mini- and microcomputers is a variety of addressing modes which require different instruction lengths in one machine. The most straightforward example occurs in the IBM 1130. IBM 1130 instructions may be either "short format," with the operand address ±127 words (actually +127 words, −128 words due to two's-complement representation; we ignore this detail in the future) from the current value of the program counter, or "long format," with an entire 16-bit word containing the full memory address of the operand. Thus a reference or branch to a "nearby" location requires a one-word instruction, while a "faraway" reference requires two words. If we consider the 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

This research was supported in part by the National Science Foundation under Grant MCS-76-17323.
Author's address: Computer Science Department, Indiana University, Bloomington, IN 47401.

differences in addressing cost. Table I compares the addressing mode variations of a number of machines.

When generating code for machines with different address modes, it is of course desirable to use the shorter formats whenever possible. This use not only decreases the program space requirement, but also decreases 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 principal activity in the latter case would be moving variables to locations "nearby" to 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—the simplest one available—has two addressing formats, which we call "short" and "long." All instructions are single address, with a short instruction (i.e. an instruction whose address is in the short format) occupying 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 only refer to operands within up to a fixed distance l forward or backward from the

Table I. Addressing Mode Variations

Machine	Nearby	Faraway
Motorola 6800	Bytes 0 through 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 address 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, indirection, etc.)
CDC 6600 PPU	PC ± 31 words (branches only)	12-bit address
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 registers pointing to 4096-byte blocks	Must load a new base register value
Digital Equipment VAX-11	Branch to PC ± 127, operands within 8-bit displacement of PC or other base registers	Branch to PC ± 32767, operands with 8- or 16-bit displacement

* PC indicates the current value of the program counter, which contains the location of the instruction to be fetched.

instruction containing it. In the following, "l" is used consistently for this hardware-related parameter. Although this is a simple model, it corresponds quite closely to an IBM 1130.

The notation for this model consists of identifying labels with data words and with instructions and their operands, so that $\alpha : \beta$ represents an instruction labeled α which has the word labeled β as an operand. Note that $\alpha : \beta$ is an affirmative statement about the instruction labeled α. Of course, each instruction has a unique label but a label may occur in many operands. The *distance* between two instructions or 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 time dependent, since the number of words between two instructions depends upon whether intervening instructions are long or short. We assume that initially all instructions are short; thus the *initial distance* will be 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 $\alpha : \beta$ occurs between $\gamma : \delta$ and the operand δ. 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 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: $\alpha : \beta$ and $\beta : \gamma$ are instructions, $\alpha : \beta$ is in the span of $\beta : \gamma$, $d(\alpha, \gamma) = 1$, 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 (that is, they are data words or instructions whose operands are guaranteed to be less than l or to be greater than l words away). The instruction $\beta : \gamma$ is dependent on $\alpha : \beta$.

2. IN-PLACE CODE OPTIMIZATION

The simplest type of optimization problem on a short/long address machine is to choose exactly those instructions which must have long addresses in an otherwise fixed piece of source code. We call this *in-place optimization*. This optimization could be performed by an assembler. It is clear that such an optimum uniquely exists: if the order 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. The initial distance is defined with all instructions short with such cases in mind. The following

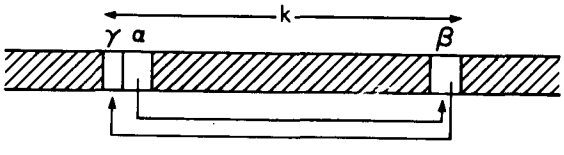


Fig. 1

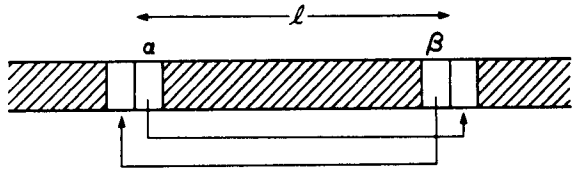


Fig. 2. Two instructions which must both be the same length

algorithm will not expand such situations unnecessarily, but detecting them in order to collapse them would be difficult.

A simple algorithm, which provides the best possible in-place optimization, although not in the most efficient manner, 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. Unfortunately, this algorithm may require $O(n)$ passes through the program if instruction references are chained in a pathological way. A number of systems, however, have made use of this multipass algorithm. The BLISS compiler uses this technique, along with other methods, to optimize branch instructions on the PDP-11 [15, p. 119]. A variation of this algorithm, which misses the optimum in cases such as in Figure 2, starts with all instructions long and shortens as many instructions as possible during each pass. Using this variation, 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 [2, p. 53]. The UNIX assembler makes three passes. Linear algebraic solutions, of high computational cost, are presented in [3] and [10].

There is, however, a relatively simple algorithm which performs in-place optimization in linear time. This technique makes one initial pass through the program gathering two kinds of information: a table containing the (initial) distance of each instruction, and a graph linking each instruction to those others which depend upon it (recall $\gamma:\delta$ depends on $\alpha:\beta$ if $\alpha:\beta$ is between γ and δ). At the end of this pass each instruction known (or discovered) to require long format is processed by incrementing the distance of all its dependent instructions. A similar algorithm is given by Szymanski [14].

This crude sketch does not specify how the dependency information can easily be obtained in a single pass, and it could produce an enormous data structure, with $O(n \cdot l)$ entries, if implemented directly. However, a number of heuristics produce a reasonable algorithm:

(1) Since it is unnecessary to place an instruction on dependency lists if it requires long format, the pass through the data may be made with a "window" (buffer) of size l.

(2) The initial pass could be incorporated with a first pass of an assembler, and the length of all instructions could be computed prior to the second pass, along with a table of address "patches."

(3) A double window could handle most local references before they were entered in the dependency structure.

(4) Two counters could keep track of the maximum and minimum distance for

each instruction, and an instruction would be known to be short if its maximum distance fell to l or below.

(5) As soon as an instruction was known to be long or short, the counts of all instructions which depend upon it could be incremented or decremented, respectively.

In general, these heuristics process as much dependency information as possible "on the fly," reducing the size of the data structure. An algorithm incorporating these and other heuristics is given in [11].

Szymanski also notes that the algorithm will not work if operands of the form

$$\text{label} \pm \text{constant}$$

are allowed. The reason is that lengthening one instruction may cause another to be closer to rather than farther from its operand. Indeed, Szymanski shows that problem of optimizing code with such pathologies is NP-complete. However, a necessary condition for these difficulties to arise is that an operand is, say, $L + C$, and one at least of the C instructions following L is variable format. Specifying an operand by $L + C$ in such a case is bad coding, to say the least, and an assembler should flag such usage as an error.

3. OPTIMIZATION WITH STORAGE ALLOCATION IS HARD

Knowing that it is easy to optimize a program if no rearranging of storage is permitted, it is natural to ask whether arranging 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. This optimization may be inappropriate or undesirable in certain situations. Placing data and code together violates the tenets of "pure" code, making reentrant/recursive programming difficult or impossible. There are, however, occasions where impure code is appropriate, as in a systems programming language for a single-user minicomputer system [6, 12]. We use *storage allocation optimization* to denote the process of arranging modules and data objects so as to minimize the number of long-format instructions.

We consider that there are certain (one-word) variables whose location in the program body is not fixed a priori and certain *slots* (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 an unallocated variable will be indicated by an arrow pointing directly at a label (rather than a cell) and a slot by a heavy vertical bar (see Figure 3). 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 in a sense which has already become classic in its brief lifetime: the problem is NP-complete [1, 4, 8]. The NP-complete problems include many from

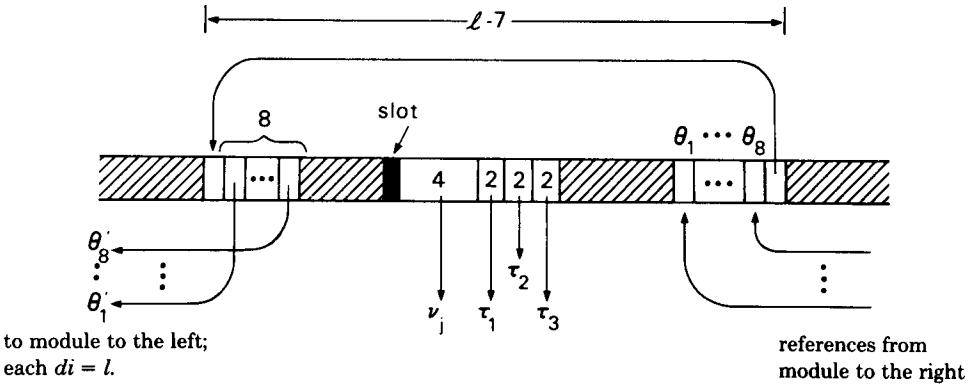


Fig. 3. The t-module corresponding to V_j

combinatorics, operations research, and other areas whose best-known algorithms require exponential time. They are all related in that the discovery of 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.

A problem is *NP-complete* if (1) an instance of the problem may be solved nondeterministically in polynomial time, and (2) some standard problem (known to be NP-complete) may be reduced to the given problem. 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 with domain S. $\langle S, P\rangle$ reduces to $\langle T, Q\rangle$ if there is a translation $\mathrm{tr} : S \rightarrow T$, which operates deterministically in polynomial time, such that, for each $x \in S$, $P(x)$ iff $Q(\mathrm{tr}(x))$. Because a more complicated formulation is required we ignore condition (1) in the following (technically proving the problems "NP-hard" instead of "NP-complete"). The standard problem which we use in (2) is conjunctive normal form (CNF) satisfiability. Each instance F of CNF is a Boolean formula

$$C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m,$$

where each C_i is a *clause* of the form

$$A_1 \vee A_2 \vee \ldots \vee A_k,$$

and each A_j is a *literal*, that is, either V or $\sim V$ for some variable V. The problem is to determine whether there exists an assignment of *true* or *false* to each variable of F such that F evaluates *true*. Such an assignment is said to *satisfy* F.

We assume certain conditions on the formulas of CNF, which do not restrict generality in the sense that for every formula F there is an F' (of comparable size) meeting these conditions and such that F is satisfiable iff F' is. The first of these conditions, which requires that no variable appears both negated and unnegated in some clause, is trivial. The second, that each clause has at most three literals, is from Cook [1]. The third condition is that each literal appears in at most three clauses. To verify that the third condition does not restrict generality, observe that the conjuncts $(V \vee \sim W)$ and $(\sim V \vee W)$ appearing in F

force V and W to have the same value in any assignment of values satisfying F. Thus if a variable V appears in more than three conjuncts, we may replace two occurrences of V by some new variable W and introduce the new conjuncts $(V \vee \sim W)$ and $(\sim V \vee W)$. Occurrences of $\sim V$ are treated similarly. This process is iterated until no literal appears in more than three conjuncts. The resulting formula is satisfiable iff the original one was. It is no more than four times the length of the original. A formula F satisfying these conditions will subsequently be denoted $C_1 \,\&\, C_2 \,\&\, \ldots \,\&\, C_m$, where each conjunct C_i is $A_{i,1} \vee A_{i,2} \vee A_{i,3}$ (or $A_{i,1}$ or $A_{i,1} \vee A_{i,2}$, which are treated similarly and thus are not distinguished). The variables of F are $V_1, V_2, \ldots, V_n$. For each V_j, $c_{j,1}$, $c_{j,2}$, and $c_{j,3}$ (possibly fewer) denote the indices of the conjuncts in which V_j appears, and $\bar{c}_{j,1}$, $\bar{c}_{j,2}$, and $\bar{c}_{j,3}$ index conjuncts containing $\sim V_j$.

We do not prove that storage allocation optimization is NP-complete, since we shortly prove much stronger results.

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 [5, 13]. 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 one of minimization it is reasonable to ask how close can we come to the minimum with an "efficient" algorithm. A minimization problem is said to have an r-approximate algorithm $(r > 1)$ if, for each instance of the problem,

$$\$_A \leq r \cdot \$_O,$$

where $\$_A$ is the cost of the solution provided by the algorithm and $\$_O$ is the cost of the true optimal solution.

Before considering approximation algorithms for storage allocation, it is necessary to define the cost measures used in the approximation. The most natural measure is the *program size measure*, where we count the total number of words in the final object program. The *long-instruction measure* counts only the number of instructions which are in long format.

THEOREM 1. *For any $r \geq 1$, the problem of finding an r-approximation to storage allocation optimization with long-instruction measure is NP-hard. Moreover, the maximum short address span l may be chosen to be a fixed constant* (35 *or greater*).

PROOF. The proof is by reduction of CNF, with the above restrictions, to storage allocation optimization.

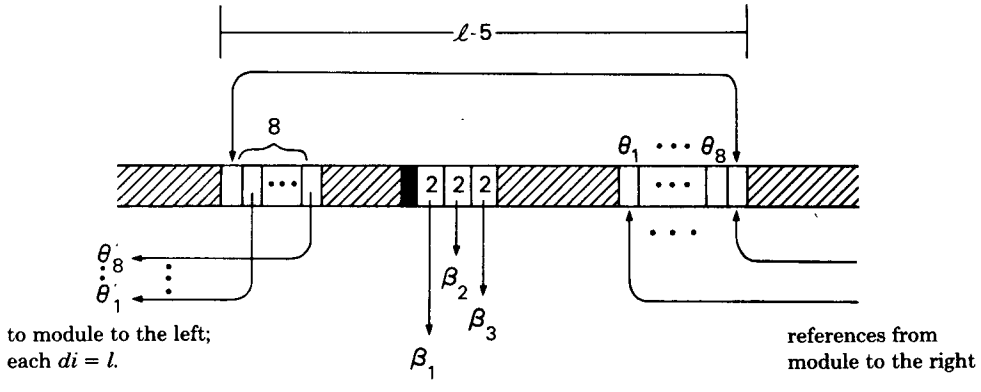
Say $r \geq 1$ is fixed and we are given a formula F with clauses C_i, $1 \leq i \leq m$, and variables V_j, $1 \leq j \leq n$. We construct a storage allocation problem, in a manner described below, such that allocation of certain program variables is related to the assignment of truth values to corresponding logical variables. If F is satisfiable, the program variables can be allocated based on a satisfying assignment of truth values while forcing relatively few instructions to long form. However, any allocation of program variables which does not correspond to a satisfying assignment forces a large (dependent on r) number of instructions to be long format. Thus any allocation which even approximates the optimal must correspond to a satisfying assignment, if indeed F is satisfiable. If F is not satisfiable, all allocations of program variables force most instructions into long format.

The storage allocation problem associated with F is constructed with one module, called a "c-module," for each C_i and two modules for each V_j. The modules for V_j are called the "t-module" and the "f-module," since the allocation of an associated program variable (v_j) to the t- or the f-module corresponds to assigning, respectively, *true* or *false* to V_j. Figure 3 shows the t-module corresponding to V_j; the f-module is constructed similarly except that $c_{j,s}$ in the subscripts of the α's is replaced by $\bar{c}_{j,s}$. The c-module corresponding to C_i is shown in Figure 4. The unallocated program variables of the code problem correspond to the variables of the formula and to the occurrences of logical variables in clauses. In particular, there is a program variable labeled v_j for each V_j and a program variable $\alpha_{j,i}$ for each clause C_i in which V_j appears.

The modules are 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 intermodule instruction is exactly l. This condition is achieved by having a sufficient number of labels (maximum 8) 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.

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 t- or f-module may be short form only if at most seven of the instructions within its span are long form. The check instruction is forced to be long either because labels referenced in an adjacent module have been "moved" too far away or because certain variables have not been allocated to the module, thus forcing the instructions referencing them to be long. The first of these conditions, which force the check instruction of a given



where β_s is $\alpha_{j,i}$ if $A_{i,s}$ is V_j or $\sim V_j$, and the θ's are labels unique to each module

Fig. 4. The c-module for clause C_j

module to be long, involves the eight instructions (the *link* instructions) at the left of the module that reference labels in the next module. The only significant dependence of the link instructions (ignoring dependence on other link instructions or on instructions whose length is fixed) is on the check instruction of the next module to the left. Since the initial distance of each link instruction is l, they all can be short if and only if the check instruction of the next-left module is short. If the next-left check instruction is long, then all the link instructions must be long, which in turn forces the check instruction of the given module to be long. In this way, lengthening one check instruction propagates to lengthening the rightmost check instruction.

The second condition forcing the check instruction to be long arises if insufficient program variables are allocated to the module. Consider the t-module for V_j, as illustrated in Figure 3, and assume the check instruction of this module has not already been forced long by the link instructions. Then the maximum cost (the number of additional words required for data variables plus long-format instructions) is seven if v_j is allocated to the module. This cost comes from the one word allocated for v_j plus six words required if all of the instructions referencing α's (see Figure 3) are long format. On the other hand, if v_j is not allocated to the module, the least cost is seven, including four words for the instructions referencing v_j and one word allocated for each of the α's. Thus the check instruction can remain short if either the operand v_j or the α operands are assigned to the module. Since both the t-module and the f-module of V_j reference v_j, and since the intermodule filler of length l prevents v_j from being near both modules, one of the modules must be assigned v_j while the other must be assigned its corresponding α's, if we are to avoid exploding the check instructions. Say v_j is assigned to the t-module, then $\alpha_{j,i}$ is available for the c-module of those C_i's in which V_j appears as a literal. But these are exactly the clauses which are satisfied when V_j is assigned *true*. Similarly, if v_j is allocated to the corresponding f-module, then the α's are available for any c-module corresponding to a C_i satisfied when V_j is *false*.

A similar analysis shows that the c-module for C_i must have assigned to it at least one operand $\alpha_{i,j}$ where V_j or $\sim V_j$ appears in C_i.

To sum up the construction so far, the assignment of v_j to the corresponding t- or f-module corresponds to an assignment of *true* or *false*, respectively, to V_j, 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 add a block of $q + 1$ instructions (the exact value of q is given later) labeled $\gamma_0, \gamma_1, \ldots, \gamma_q$ (Figure 5). If θ_8 is the indicated label from the rightmost

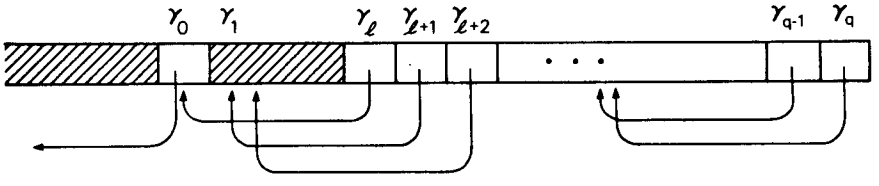


Fig. 5. Block of γ at end of program (each initial distance is exactly l)

module, then $\gamma_0{:}\theta_8$. Enough filler is provided so that $di(\gamma_0, \theta_8) = l$. The $l - 1$ instructions $\gamma_1, \ldots, \gamma_{l-1}$ have no operands and $\gamma_s{:}\gamma_{s-l}$ for $l \leq s \leq q$. The placement and construction of this block is such that γ_0 is forced long iff the rightmost check instruction is forced long; and forcing γ_0 long forces each of $\gamma_l, \ldots, \gamma_q$ to be long in turn. Thus each of these $q - l$ 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

$$q > p \cdot r + l.$$

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

$$\$_A \geq q - l > p \cdot r \geq \$_O \cdot r.$$

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

Examining Figure 3, we see that the f- and t-modules, which are larger than the c-modules, may have an initial length as short as 26, and hence an l of 35 suffices. □

THEOREM 2. *For any r, $1 \leq r < 2$, the problem of finding r-approximate solutions to storage allocation optimization with program size measure is NP-hard. Moreover, l may be any fixed integer greater than 34. However, a 2-approximate solution to this problem is trivial.*

PROOF. Perform the construction exactly as in Theorem 1, 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

$$q > (l + 2(r - 1)p)/(2 - r).$$

Since $1 \leq r < 2$, $2 - r$ is always positive so this restriction is equivalent to

$$q > l + p + (r - 1)(2p + q).$$

Now assume F is satisfiable. Then, counting one word for each γ which is short but allowing two words for each of the other p instructions, we have

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

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

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

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

As noted above, $\$_A \leq 2 \cdot \$_O$ from the nature of the problem. The bound of Theorem 1, $l \geq 35$, again suffices. □

The above constructions required specified "slots," which are the only locations

to which variables may be allocated. These slots are used to simplify proofs which are already messy enough. However, if the modules are arranged such that no two adjacent modules reference the same variables (which may be easily done since no two modules of the same type share variables), then the presence of intermodule filler is sufficient to complete the above proofs without the requirement that variables be placed only in slots.

4. HEURISTIC CONSIDERATIONS

Section 3 demonstrated that even approximation of the problem of code generation with storage allocation 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 variables during compilation or assembly and with 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 commonplace programming technique, which is certainly economical in Fortran on a machine with fixed address format, is to declare a few temporary variables which are used repeatedly throughout a program. However, in machines with instructions of short/long format, this is potentially quite wasteful. Local variables should be used as much as possible, within small local blocks if the language permits.

A corollary to the programmer heuristic for local variables is a recommendation to place loop indices within or adjacent to loops. Many **for**-loop structures have an unconditional branch, and the loop index may be placed immediately following this branch. This may be done by the programmer in assembly language or by the complier in a language (like Algol W, except that Algol W is recursive) which implicitly declares a loop index for the body of the loop.

Automatic heuristics, applied by an assembler or compiler, presuppose the freedom to rearrange storage. The natural "greedy" heuristic counts the number of times each variable is referenced by each module and associates the variable with that module which references it most frequently. Considerations of branch instructions are omitted. 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. 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 3). However, the heuristic can be implemented without much additional overhead as part of a first pass.

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. 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. Alternatively, a notation for label temporaries (comparable to the "3F" and "7B" labels in MIX [9]) 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.

Consideration of block structured but nonrecursive languages demonstrates 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 $\pm l$ addressing range restricts the useful size of blocks with local variables. The usual block structure restricts declarations to the beginning of blocks, and it is natural for compilers to allocate storage immediately. This means that data references never make use of the forward range of short instructions. Three solutions to this restriction seem to exist. One lies with the compiler, at the expense of considerable complication and perhaps another pass. This solution is to allocate storage in the middle of the block. The second solution is architectural and not out of the question for microprogrammed machines. This is to have short data references in the range $-2l$ to 0, perhaps retaining the $-l$ to $+l$ range for branches. Finally, short data references may be displacement from a base address (for example, the Data General Nova has displacement from base registers as well as PC-relative addressing modes). Unfortunately this architecture does not make storage allocation easier. The NP-completeness results of Section 3 carry over without much effort: all base registers but one must (on penalty of exploding check instructions) reference certain global blocks, and each module of the construction must share certain variables with two new modules, so none of the original modules may overlap data areas.

Another observation about short/long addresses is that indirection (as in the PDP-8) is better than an extra word with a long address. Indirection within the $\pm l$ 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 segment 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 in 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 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 are replicated to shorten instructions and

thus save execution time, but where the replication uses more space than is saved by short instruction formats.

ACKNOWLEDGMENTS

The author thanks D.S. Johnson, F.R. Keller, and J.L. Peterson for comments on this material.

REFERENCES

1. COOK, S.A. The complexity of theorem proving procedures. Proc. 3rd Annu. ACM Symp. on Theory of Computing, May 1970, pp. 151–158.
2. Common assembler language user's manual. Interdata Corp., Oceanport, N.J., 1974.
3. FRIEDER, G., AND SAAL, H.J. A process for the determination of addresses in variable length addressing. *Comm. ACM 19*, 6 (June 1976), 335–338.
4. GAREY, M.R., AND JOHNSON, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, 1979.
5. GAREY, M.R., AND JOHNSON, D.S. Performance guarantees for heuristic algorithms. In *Algorithms and Complexity: New Directions and Results*, J.F. Traub, Ed., Academic Press, New York, 1976, pp. 41–52.
6. HARRIS, K. The design and implementation of a compiler for a mini-computer. M.S. paper, Comptr. Sci. Dept., Pennsylvania State U., University Park, Pa., Aug. 1976.
7. JOHNSON, D.S. Approximation algorithms for combinatorial problems. *J. Comptr. Syst. Sci. 9*, 3 (Dec. 1974), 256–278.
8. KARP, R.M. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, Miller and Thatcher, Eds., Plenum Press, New York, 1972.
9. KNUTH, D.A. *The Art of Computer Programming, Vol. I: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.
10. RICHARDS, D.L. How to keep addresses short. *Comm. ACM 14*, 5 (May 1971), 346–349.
11. ROBERTSON, E.L. Code generation for short/long address machines. Tech. Summ. Rep. 1979, Math. Res. Ctr., U. of Wisconsin, Madison, Wis., Aug. 1977.
12. ROCHE, P. Code generation for PL 1600. M.S. paper, Comptr. Sci. Dept., Pennsylvania State U., University Park, Pa., Feb. 1977.
13. SAHNI, S., AND GONZALEZ, T. P-complete approximation problems. *J. ACM 23*, 3 (July 1976), 555–565.
14. SZYMANSKI, T.G. Assembling code for machines with span-dependent instructions. *Comm. ACM 21*, 4 (April 1978), 300–308.
15. WULF, W., JOHNSON, R.K., WEINSTOCK, C.B., HOBBS, S.O., AND GESCHKE, C.M. *The Design of an Optimizing Compiler.* American Elsevier, New York, 1975.

Received October 1977; revised June 1978