Programming Techniques
G. Manacher, S.L. Graham Editors

A Process for the Determination of Addresses in Variable Length Addressing

Gideon Frieder and Harry J. Saal
IBM Israel Scientific Center, Haifa

An algorithm is presented for the assignment of instruction addresses and formats under the following conditions: (1) the length of the instruction varies as a function of the distance of the instruction from its target; (2) there exists an optimality criterion which implies some preferential choices subject to the addressing constraints. This may be, for example, achieving the smallest number of long instructions, in which case the total code length is minimized, or minimizing the assigned address of a specified point in the program. The algorithm is suitable for arbitrary program structure and a choice of optimization criteria.

Key Words and Phrases: variable length addressing, assembler, paging
CR Categories: 4.11, 4.12

Authors' addresses: G. Frieder, Computer Science Department, SUNY at Buffalo, 4226 Ridge Lea Road, Amherst, NY 14226; H. J. Saal, Israel Scientific Center, IBM Israel Ltd., Computer Science Building, Technion City, Haifa, Israel.

1. Introduction

In a recent paper, D.L. Richards [1] proposes a solution to the following problem: given two relative addressing modes, long and short, and given the forward and backward distances which are the thresholds for the choice of the mode, assign proper long or short addresses so that the total length of the code is minimal.

In his work, Richards introduces an algorithm which finds an assignment for the addressing modes. However, his solution is not always optimal, as can be seen from the failure of his algorithm to produce a minimal length assignment in a simple case (see Appendix), and the algorithm may fail completely in some pathological cases (discussed later). Apart from these deficiencies, there are other drawbacks to the algorithm, among them the necessity to distinguish between inter- and intra-block references and the rigidity of the optimization criterion.

The present work uses, in part, the ideas introduced in [1]. We present an algorithm which handles code segment structures in a general fashion, and can use other optimization criteria. Our algorithm can be adapted for address assignment in a paged environment where in-page and out-of-page addresses are of differing lengths, a common situation in minicomputers and microprocessors [2].

The actual code for the algorithm can be found in [2], together with data and results for three experimental runs. We shall comment on the applicability of the algorithm to practical problems in a later section.

2. Basic Assumptions and Data Description

We assume that the two addressing modes differ by length d. The short instructions (those with the short addresses) can be used whenever the target, i.e. the address of their operand, is within the distance F forwards or B backwards from the instruction. In each program, some of the mode decisions can be made trivially during the first pass through the program, such as some backwards references. The unresolved instructions and their targets will be denoted by the index set $1, 2, \ldots, N$. The structure of the program is described by a binary matrix C, where $C_{ij} = 1$ if and only if the jth instruction is encountered when tracing a path from the point to which all addresses are relative to the ith instruction. There are, therefore, entries for every unresolved instruction which directly affects the location of instruction i. We do not attempt optimizations which reorder the code segment sequences, possibly introducing additional branches. Such modifications could be advantageous (as in optimizing compilers), but are inappropriate for the low-level address assignment phase we are considering.

The targets of the instructions are described by the vector G, whose ith component is the index of the

target of the ith instruction. Both C and G can be computed directly from the given program. They serve as data to the algorithm, which will proceed without any further reference to the structure of the program.

Note that programs tend to cluster their references into two distinct groups: (1) near references, reflecting loops, targets of conditional branches, and other local phenomena, and (2) long-range references, reflecting data accesses, intra-block references, and procedure calls [6]. Some parts of these references are trivially resolved and will, therefore, not appear in C or G. This tends to hold the actual problem to a manageable size.

3. The Algorithm

From the data extracted from the program in question, i.e. from the matrix C and the vector G, we first compute some auxiliary variables. Let $\hat{P}$ be a vector whose ith component is the value of the ith unresolved address or target under the assumption that all addresses are chosen short. Let Q be a binary vector such that $Q_i = 1$ if the ith entry is an unresolved address reference, and L be a vector so that $L_i = d$ when the ith instruction is long, and zero otherwise. Initially all components of L are zero, and all components of Q are 1.

One should note that for any choice of L, the address values P are given by

$$P_i = \hat{P}_i + \sum_j C_{ij}L_j .$$

The relative distance between an instruction and its target can be written as

$$R_i = \hat{R}_i + \sum_j D_{ij}L_j$$

where $\hat{R}$ is the vector of minimum relative distances

$$\hat{R}_i = \hat{P}_{G_i} - P_i$$

and where D is a ternary matrix having elements -1, 0, 1 so that the ith row of D has a -1 in each element that influences the position of i and a $+1$ in each element that influences the target of i. The dependency matrix is calculated by

$$D_{ij} = C_{G_i j} - C_{ij} .$$

In the first and preparatory part of the algorithm, we compute $\hat{P}$, $\hat{R}$, D and two limiting vectors E^1 and E^2 so that

$$E_i^1 = \hat{R}_i + d \sum_j D_{ij}\delta(D_{ij} = 1)$$

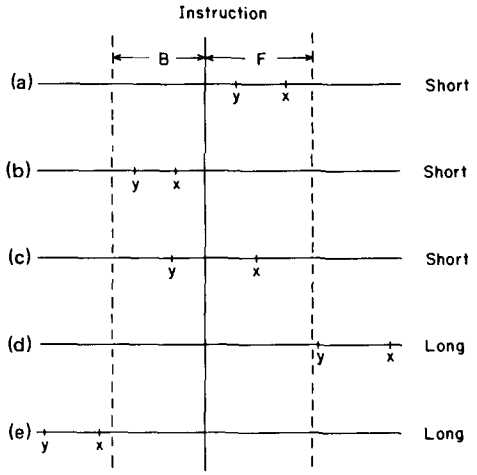
is the maximum separation, and

$$E_i^2 = \hat{R}_i - d \sum_j D_{ij}\delta(D_{ij} = -1)$$

is the minimum separation between the instruction and its target. $\delta(s = t)$ has the value 1 if $s = t$ and the value zero otherwise.

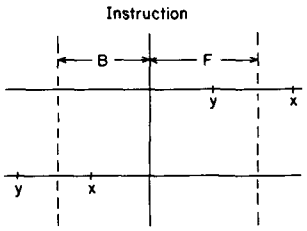
A graphical representation of the various possible relations between an instruction and its target which also exemplifies the meaning of E^k is presented in Figures 1, 2, and 3. If the instruction in question is taken as the origin and the values of E^k are as described in Figure 1, its address mode may be chosen short (cases a, b, c) or must be long (cases d, e). Such a situation will be labeled as "simple."

Fig. 1. Simple cases: $y = E^1, x = E^2$.



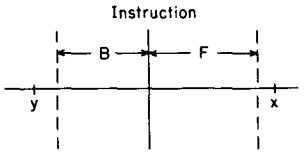
There are, however, more complicated possibilities in which the decision on the mode cannot be made (Figure 2):

Fig. 2. Undecided cases: $y = E^1, x = E^2$.



or even where the direction of the branch is unknown (Figure 3):

Fig. 3. Not-well-ordered cases: $y = E^1, x = E^2$.



The latter case, which we shall call not-well-ordered, was not recognized in the previously referenced work [1]. Note also that the procedure presented here deals in a uniform way with both inter- and intra-block references.

The next step in the algorithm is to resolve all

simple cases. These are detected by locating all E_i^k in which both components, i.e. both Ei^1 and E_i^2, are either inside or outside the B–F range (see Figure 1). In these cases, the instruction modes are fixed, and both Q and L are properly updated. Based on these new values, R and E are recomputed and the search for further simple cases is repeated. This has to be done because the instructions that have just been resolved may change some undecided cases into simple ones. This process is terminated when all simple cases are exhausted. Note that in this process we already included some assumptions about the "optimality" criterion since all instructions which *had to be* long were so chosen, but we chose as short all instructions which *could have been* such. This is not necessarily beneficial for arbitrary optimization criteria. We shall defer to a later point the discussion of such criteria.

When all simple cases were exhausted, the algorithm proceeds, as suggested in [1], to compute the threshold vector T. This vector is defined so that T_i is the distance of each address from being assigned as short, assuming that all further addresses are short. One should note that this calculation is not always possible, as cases may arise (Figure 3) in which both the mode and the direction of the target cannot be decided. A test is therefore performed to find all these pathological cases and they are assigned long addresses. Any new simple cases are then handled as described above.

At this point we are left with all undecided cases (Figure 2) in which the direction is determined. The threshold vector is computed. If all entries are zero or less, a solution exists in which all addresses can be chosen short. Otherwise some have to be long. The dependency matrix is now changed by inverting the signs of all entries describing backward references. After this change, $\sum_j D_{ij}L_j$ expresses the contributions towards achieving the threshold for either backward or forward references for instruction i.

We now consider possible choices of L. The choice procedure sequencing is governed by a criterion function. For example, this function can choose the minimum number of long instructions (the case in [1]), or may be more complicated, such as the minimization of the location of a certain reference and the minimization of the remaining code length (see [2], example 3).

We do not suggest exhaustive generation of all possible combinations of the components of L as an optimal means of locating a solution, although in [2] we employ such a procedure. For example, one can first order the still unresolved part of the dependency matrix into block diagonal form, which exposes independent subproblems, each of which can be solved separately. This avoids an exponential explosion of the number of possibilities to be tested in many practical cases, particularly in the adaptation of this algorithm to addressing in paged memories [2]. Observe that by adding suitable positive entries to the diagonal elements of D, we produce a problem of set-covering or zero-one programming with linear constraints. The literature is so rich on the solution of these problems that we reference only three recent papers [3, 4, 5] from which most others can be located.

For each choice, $\sum_j D_{ij}L_j$ is recalculated and compared to T_i for all those addresses which we want to keep short. Whenever there is a match, i.e. all components agree, one has an acceptable solution, and the actual addresses are computed by

$$P_i = \hat{P}_i + \sum_j C_{ij} L_j$$

Two remarks are now in order. The first one applies to the way that criterion functions are applied for simple cases. In those cases, when the criterion is different from merely minimal code length, R and E calculations are changed so that only those instructions which have to be long will be so set and the others left intact. Their mode will be determined during the threshold matching part of the algorithm, so that the criterion function will be in effect.

The second remark concerns the application of the algorithm for the paged case. In [2] the present authors outlined an extension of the present algorithm to the paged case, with the observation that there is a noted similarity between these two problems. We do not reproduce the extension here, in order to keep this note as short as possible.

4. Conclusions and Closing Remarks

In [2] we present the relative addressing algorithm coded in APL, complete with examples. As pointed out in this presentation, the algorithm generalizes and corrects the approach in [1]. We believe that in the form presented it is very easy to use. Part of the clarity and brevity of the algorithm is derived from the use of a proper mathematical notation for arrays provided by APL. For the benefit of those not conversant with APL, we have used conventional mathematical notation in this paper.

There are certain generalizations possible for the algorithm, as pointed out by Richards. We believe that the algorithm in the way presented here is suitable for generalizations in more complicated cases, such as multiple instruction lengths or page addressing. Our development was triggered by the need for a working algorithm for a high-level assembler, in which there were only two modes. We therefore did not try to develop the most general case.

Appendix

The algorithm that was introduced in [1] produces a nonoptimal solution for the following example, whereas our procedure produces the correct result.

The example consists of three unresolved addresses, all of which must be short or long simultaneously.

The program has the following structure:

inst. no	label	target
0	K	B
	⋮	
7	A	C
	⋮	
15	B	K
	⋮	
22	C	A

All intervening instructions are fixed in length.

The structure of the program is expressed by the matrix C and the vector G which have the values

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}, \quad G = (3 \quad 4 \quad 1 \quad 4).$$

The "all short" address assignment, which is the starting point of the calculation, is given by $\hat{P} = (0\ 7\ 15\ 22)$. With the values of $B = F = 15$ and $d = 1$, one can proceed with the calculation as outlined in the paper. The final result is given by the values of L and P, which in our case are $P = (0\ 7\ 15\ 22)$, with all components of L being zero. This means that all addresses were assigned short. As pointed out before, a calculation by the procedure outlined in [1] yields all addresses long.

Received January 1974; revised May 1975

References

1. Richards, D.L. How to keep the addresses short. *Comm. ACM 14*, 5 (May 1971), 346–349.
2. Frieder, G., and Saal, H.J. On the determination of addresses in variable length addressing. Report 9, IBM Israel Scientific Center, Haifa, Israel, 1974.
3. Balas, E., and Padberg, M. On the set covering problem. *Op. Res. 20*, 6 (June 1972), 1152–1161.
4. Salkin, H.M., and Koncal, R.D. Set covering by an all integer algorithm: computational experience. *J. ACM 20*, 2 (Feb. 1973), 189–193.
5. Christofides, N. Zero-one programming using non-binary tree search. *Computer J. 14*, 4 (Apr. 1971), 418–421.
6. Saal, H.J., and Shustek, L.J. On measuring computer systems by microprogramming. State of the Art Report on Microprogramming and System Architecture, INFOTECH, Maidenhead, Berkshire, England, 1975.

Programming Techniques

G. Manacher, S.L. Graham Editors

Referencing Lists by an Edge

David S. Wise
Indiana University

An edge reference into a list structure is a pair of pointers to adjacent nodes. Such a reference often requires little additional space, but its use can yield efficient algorithms. For instance, a circular link between the ends of a list is redundant if the list is always referenced by that edge, and list traversal is easier when that link is null. Edge references also allow threading of nonrecursive lists, can replace some header cells, and enhance the famous exclusive-or trick to double-link lists.

Key Words and Phrases: list processing, circular, doubly linked, overlapping sublist, header cell, pointer, cursor

CR Categories: 3.73, 4.10, 4.22

Introduction and Definition

The purpose of this paper is to indicate some natural advantages of referring to and into lists by pointing to an edge (i.e. to two adjacent nodes). These appear as space savings for data structures, and often as time savings for algorithms which depend upon edge references.

In the following sections the list structures under consideration are defined, and the technique of referring *to* these structures by the edge between first and final nodes is demonstrated. The sample algorithms, not

Author's address: Computer Science Department, Indiana University, Bloomington, IN 47401.