Automatic segmentation of programs for a two-level store computer

By F. H. Dearnley and G. B. Newell*

This paper is based on a similar paper of the same title presented at the Joint Computer Conference held in Edinburgh 31 March–3 April 1964. It presents a new algorithm for the segmentation of programs produced for a two-level store computer. This algorithm is efficient both in its operation and in the results it produces. The presentation takes the form of a description of the program which applies the algorithm.

Introduction

The use of two-level stores as computer memory introduces extra programming effort and some loss of efficiency due to the inter-level transfer. The loss of efficiency can be combated by the use of autonomous transfers. This paper presents a software solution to the former problem, which allows the programmer to treat the machine as if it had a single-level program store. This solution is an efficient automatic segmentation scheme.

Clearly such a scheme should not exist in isolation, but should form the final phase of a compiler or assembler. Under such considerations the segmentation scheme has been designed to accept programs in the form of a string of pseudo-orders backed by some tables. This is the normal form of intermediate stage produced by compilers and assemblers.

For the segmentation scheme to be acceptable it is of major importance that its object programs have running times comparable to those of the equivalent programs produced (and segmented) manually. Furthermore, for the segmentation process to be used within an assembler —particularly within a "load and go" assembler, it is important that the process be economical in its use of time and computer storage. The scheme satisfies the former criterion since all object program loops which can fit within a single block of program are so contained. The latter criterion is also satisfied since the process is one-pass and requires only a limited portion of the source program to be in the store at any one time.

Factors affecting segmentation

The task of segmenting programs is similar to that of batching data on magnetic tape—with the customary single exception. This exception is manifested in the treatment afforded to the jump instructions. Otherwise, logically connected units are packed together as tightly as possible into the program blocks in order to maximize the average number of object-program instructions obeyed between transfers of program blocks from the backing store.

The major consideration affecting object-program efficiency is that program loops should not be divided across blocks. Obviously this criterion cannot be applied to loops comprising more instructions than will fit into a single block. (Occasionally hand coding can score over automatic coding since such a large loop may be rewritten as two small loops, each of which can be contained in a single block. However, such a complex operation cannot be undertaken by an automatic process.) On the other hand a process which automatically starts a new program block at the head of each loop cannot claim to produce an optimal result, since in many cases the loop can be wholly contained in the space remaining in the previous block.

The scheme to be described enables an optimal object program to be produced, since the source program contains special markers interspersed in the string of pseudo-orders. These markers bracket the program loops (and any other sections of program which are required to be forced into the same object program block) and direct the scheme to start a new block only if the bracketed section overspills the current block. Since they may be nested it is essential to have two types of marker—a left-hand bracket and a right-hand bracket. The source of the brackets is left entirely to the design of the compiler or assembler using the segmentation process. Certainly with assemblers they will be direct translations of markers supplied by the source programmer. In complex compilers they may be inserted automatically after analysis of the loop structure of the source program. In addition to the brackets two other types of special marker are used. One is the end-of-program marker terminating the string of pseudo-orders. The other type is the label marker: this contains a numerical identification of the original label name within the marker. The label markers appear in the pseudo-order string immediately preceding the labelled pseudo-orders. Other pseudo-orders referring to the labels contain the numerical identification in the address bits.

It is interesting to note that forward jump instructions have no effect on the selection of segmentation points. For a forward jump to be significant it must be wholly contained within a loop; and since the segmentation process forces loops into a single block, wherever possible, the forward jump is ignorable. In the case of a loop spreading over more than one block it is immaterial whether or not a segmentation point occurs within the range of the forward jump—unless rearrangement of the source program can occur. It was decided that re-

* *I.C.T. Ltd., Research Department, 2 Gayton Road, Harrow, Middlesex.*

arrangement of program was beyond the scope of the segmentation process. Although forward jumps do not affect the selection of segmentation points, they complicate the segmentation process. When a forward jump is met it is not known if the object label will fall within the same block. The number of object-program instructions required to effect the forward jump varies according to the answer. For an out-of-block jump the object program must transfer the new program block from the backing store in addition to the jump.

A simple solution to the difficulty is to allow the maximum in every case and fill in with null orders when the label appears in-block. This solution is unsatisfactory since it results in loose block packing and hence not the optimal result. (Experience has shown that an average saving of from 15 to 20 per cent results when the aforementioned loose block packing technique is replaced by segmentation technique which produces tightly packed blocks. This does not mean that the former technique wastes an average of 15 per cent of the object-program blocks with null orders—since a form of iterative saving occurs. By packing blocks tightly the average number of labels per block increases. This results in further savings due to an increase in the number of within-block forward jumps. This enables tighter block packing, which increases the average number of labels per block; and so on to a limit.)

External appearance of the segmentation program

The segmentation program has been implemented on the I.C.T. 1301 computer. Since it was expected to form part of several compiler and assembly systems it was designed as a "black box" package. This package has six entry points which can be used by the compiler or assembler. The purpose of each entry point is as follows.

(1) The package is primed with a constant which is to appear in the same object-program block as the next instruction and will be used by that instruction. The package generates an address for the constant and puts this out for use by the compiler or assembler in constructing the next pseudo-order. The package ensures that duplications of the same constant, within a single object-program block, are removed.

(2) The package is primed with the name of a subroutine and the size of the subroutine. This entry is used simply to declare the existence of the subroutine to the package. The name and size are merely noted and no output results.

(3) The package is primed with a previously-declared subroutine name. This subroutine is to be incorporated in the same object program block as the next instruction, and will be used by that instruction. The package generates an address for the start of the subroutine and puts this out for use in constructing the next pseudo-order. References to the same subroutine by different pseudo-orders in the same object-program block result in a single copy of the subroutine appearing in that block.

(4) The package is primed with a label name. The package notes the name and puts out the numerical identification of the label. If the same label name is presented on separate entries the same numerical identification results. The output is used by the compiler or assembler to construct the special label marker, or to construct the address part of the next pseudo-order.

(5) The package is primed with a stop name. This is noted, and a unique number is assigned to the stop name. This number is put out for use in constructing the next pseudo-order (a stop instruction). This facility enables the different stops in the object program to be distinguished, which may in turn instruct the computer operator to take different actions.

(6) The package is primed with a pseudo-order or special marker. This is incorporated into the object program (after appropriate transformation) which is either put out via a peripheral unit or stored ready for obeying when the complete object program has been formed.

The main objective of the design of the segmentation package was to obtain the best possible interface between the syntax analysis and generation operations of a variety of compilers and assemblers, and the segmentation process required to produce object programs for two-level store computers. It is interesting to note that in the design described above the task of forming and utilizing the tables necessary to back up the pseudo-order string has been removed from the compiler or assembler.

Internal operation of the segmentation program

The action of the segmentation program for the first five entry points is mainly self-evident. The key to the efficiency of the segmentation process lies in the action taken when the pseudo-orders are presented. Although the process is one-pass, for the sake of clarity it will be described as a two-pass operation, and then a brief indication will be given of the manner in which this can be adapted into a one-pass operation. Further the process will be described as producing its output via a peripheral unit, the simplification required to make it a "load and go" process being obvious.

As each pseudo-order is presented to the package it is simply stored in the next position of a buffer. This buffer is of such a size that when full the pseudo-orders it contains will always give rise to more than a single object-program block. The loading of pseudo-orders into the buffer continues until it is full, when the main body of the segmentation program is entered. The first task of the process is to determine the next segmentation point. This is achieved by examining the pseudo-orders sequentially and updating two counters: a pessimistic counter and an optimistic counter. To describe the role of the two counters let us picture a graph showing their progress. This graph is a plot of the number of pseudo-orders so far processed for the current block against the space occupied by the corresponding object program (including instructions, constants, subroutines,

etc.). There is a limit to the space available in the object-program block, and we shall represent this as a cut-off line which we shall call the "blue line". The two graphs each take the form of a step function with different increments according to whether or not the pseudo-order involves a constant, subroutine, etc., and including zero increment when a special marker is encountered. Initially the two graphs follow the same path. When a forward jump is met they diverge, the optimistic counter increasing on the assumption that the label will appear in the same block, whilst the pessimistic counter increases assuming that the label will be out-of-block. When a label is met no increment is applied to the optimistic counter, but the pessimistic counter is decreased by an amount equal to the product of n and s, where s is the space occupied by the extra instructions needed to perform an out-of-block jump, and n is the number of forward jumps to the label which have appeared so far in the current block. The pessimistic counter always crosses the blue line either before or at the same point as the optimistic counter. Once the optimistic counter crosses the blue line the updating process is discontinued and the segmentation point is determined. There is no point in continuing since the optimistic counter can only increase. On the other hand when the pessimistic counter crosses the blue line it merely indicates that a possible segmentation point has been reached. It may be that the pessimistic counter will be brought back within range by the application of a decrement when a label is met.

A segmentation point is determined when one of two criteria is satisfied. Either a left-hand bracket is met and the object program up to the corresponding right-hand bracket overspills the current block, or the pessimistic counter crosses the blue line and does not return within range. Two scratch pads are kept during the updating process. When a left-hand bracket is met details of the current position are noted on one of these; a similar note is made on the other when the pessimistic counter crosses the blue line. The former pad is scratched if the corresponding right-hand bracket is met; likewise the latter if an obliging label is met. When the updating procedure terminates, the earlier of the two possible segmentation points is taken.

Once the segmentation point has been decided the second pass transforms the pseudo-orders into machine instructions and puts these out. It is known at this stage which labels are in-block, and hence the precise number of instructions required to effect each jump order. However, it is not possible to construct the addresses of forward jumps to out-of-block labels at this stage. This and similar problems are overcome by putting out blank addresses which are filled in at object-program load time by "corrections" put out when the label is eventually met.

The major problem in adapting the two-pass process described above as a one-pass process is the fact that the number of object-program instructions resulting from a forward-jump pseudo-order is unknown when it is required to put out the forward-jump instruction. To remove the problem we introduce the concept of pseudo-constants. Let us first consider the layout of a single object-program block. This consists of machine instructions starting at the beginning of the block and terminating at some point with an out-of-block jump to the next block (which is inserted automatically by the segmentation program). Also there are constants and spaces into which the loading process slots subroutines which start at the end of the block and are allocated backwards. (Any spare space left in the block appears between the end of the instructions and the beginning of the constants/subroutines.) Thus the order of output is instructions followed by constants and subroutines. To resolve the problem of forward jumps a single jump instruction is put out corresponding to the pseudo-order. If the label turns out to be in-block this is all that is required. If the label appears out-of-block then the jump address is made that of a pseudo-constant which organizes the transfer of the new program block from the backing store and jumps into it. When the segmentation point has been decided it is known which pseudo-constants are required and which can be discarded. Naturally forward jumps in the same object program block to the same out-of-block label result in a single pseudo-constant. Whenever a possible segmentation point is reached the output of instructions is halted and future pseudo-orders are stacked until it has been decided whether or not it is the true segmentation point. If not the stacked pseudo-orders are used to produce further output for the current block; otherwise they are used to produce the initial output for the next block.