# Automatic program segmentation based on boolean connectivity*

*by* EDWARD W. VER HOEF

*Ocean Data Systems, Inc.*
Rockville, Maryland

## INTRODUCTION

The past few years have seen a significant increase in the number of computers with segmented memories, i.e., computers in which executable memory is divided into a fixed number of fixed length segments. A computer so organized can offer significant advantages over the more conventionally organized machine, the foremost advantage being that such an organization facilitates running a job with only a portion of that job in executable memory. This can be done in other computers but the determination of what portions of the program must be in executable memory at any given point in the process is then the responsibility of the person writing the program, i.e., the programmer must schedule the overlaying himself and reflect this schedule in the program.

In most segmented memory computers such decisions are made by the executive or operating system. The programmer need not even be aware of these decisions. This is made possible by the fact that in such computers, any reference from within one segment to something outside that segment (whether a fetch, store or transfer) causes a trap to the executive. The executive then determines whether the segment containing the referenced item is in executable memory. If it is, the action takes place as desired; if not, the desired segment is moved from peripheral storage to executable storage and then the action takes place. Thus the advantages of such machine organization arise from this latter action but the price of these advantages lie in the executable action required even when the desired segment is already in executable memory. Techniques have been developed to minimize this cost but it cannot be eliminated completely.[1,2,3] The main thrust of these techniques has been devoted to making the executive action as efficient as possible.

However, Lowe[4] has shown that in a heavily loaded system of this type, even a small reduction in the inter-segment activity results in a significant increase in efficiency of the system. Ramamoorthy[5,6] made the observation that the paths of possible control flow may be represented by a directed graph. Furthermore he presented a method for reduction of inter-segment references which was based on the principle that the instructions represented by a maximal strongly connected subgraph (i.e., a strongly connected subgraph that is not a proper subset of any other strongly connected subgraph) should not be split into two or more segments. The difficulty in this technique is that an entire program could be a maximal strongly connected subgraph and be larger than a segment.

This article presents an algorithm for partitioning a program into some number of pieces (called pages) such that none of the pages exceeds segment size and the number of interpage (inter-segment) references is reduced. This algorithm operates solely on connectivity and size data describing the program, data readily available from a compiler or assembler with relatively modest modification. As mentioned above, inter-page references can be a fetch or store of data or a transfer of control. For simplicity of presentation, only the latter are considered in this article. However, in the research project on which the article is based, both constant and variable data were treated in like manner.[7]

## OVERVIEW OF ALGORITHM

The basic premise of the algorithm is that one should first optimize loops and then optimize the linear portions of the program. This is simply a recognition of

* The work presented here was performed at Informatics, Inc. (Washington, D. C. Division). It was supported by the U.S. Air Force Systems Command, Research and Technology Division, Rome Air Development Center, Griffiss Air Force Base, New York, under contract F30602-68-C-0285.

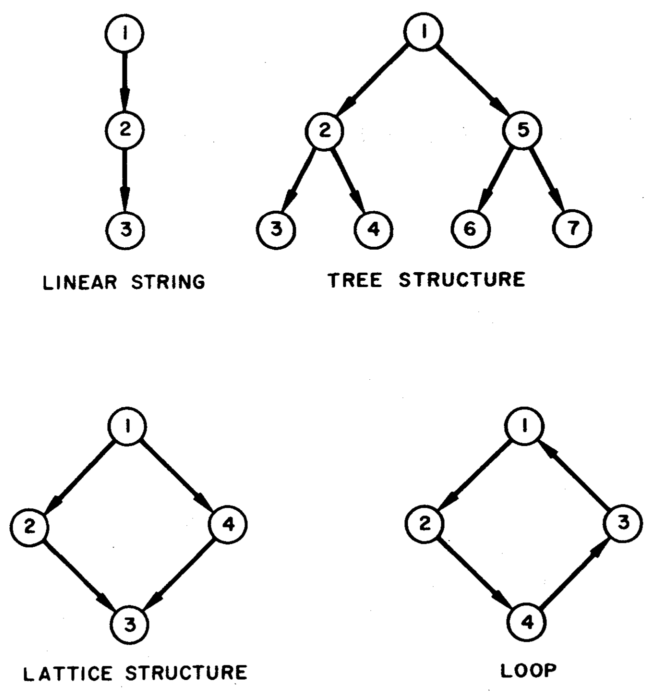

Figure 1—Types of structures

the fact that the code in a loop is executed many more times than the code in a linear portion of the program. Thus any inefficiency in a loop is paid for many times over. The algorithm operates in three phases. In phase 1, loops are detected and their component instructions are identified. The smallest loops are found first, followed by successively larger loops. The size of a loop is measured in terms of the number of program units (to be defined later) that comprise the loop. Size is a measure of the estimated time required for execution of the loop.* As soon as a loop has been detected the program units comprising the loop are merged and thereafter treated as a single unit. Phase 1 is finished after the largest loop that could fit in a single segment has been considered.

When phase 2 begins, the program consists of a collection of program units forming a linear string, a tree structure, a lattice structure, loops larger than a segment, or some combination(s) of these (see Figure 1). In this phase, a particular path is traversed and the program units along that path are identified and merged until segment size is exceeded or the end of path is encountered. In either event a new path is then selected and traversed in like manner. Phase 2 is complete when all such paths have been traversed and all merging based on connectivity accomplished. However, there may yet remain merged units of less than segment size. Such units are identified in phase 3 and merged to minimize waste space.

* Lowe[8] defines the timing estimates available at this stage of analysis as event time. He then goes on to develop the concept of unit time and shows a transformation from event time to unit time. Unit time provides a more accurate means of estimating running time than does event time but event time suffices for our purposes.

## PROGRAM DECOMPOSITION AND REPRESENTATION

The program to be analyzed is considered to consist of "units." Units are defined by Lowe[8] in the following manner. Let the smallest executable part of a program be called an instruction element, frequently consisting of a single word. An instruction unit is an ordered collection of instruction elements $e_1, e_2, \ldots e_f$ $(f \geq 2)$ such that:

1. Each instruction element of the program appears in exactly one unit;
2. For $1 \leq i < f$, the set of successors of $e_i$ consists of the single element $e_{i+1}$
3. For $1 < i \leq f$, $e_i$ is the successor of exactly one element and that element is $e_{i-1}$
4. The total volume of the unit (i.e., number of words of storage required) is not greater than some fixed maximum. (This maximum, for our purposes, will be segment size.)
5. There exists no $f' > f$ for which the above four conditions are true.

A special case is made for any element which has multiple predecessors and multiple successors and whose volume is less than the fixed maximum. Such an element is considered to be a unit.

Lowe has suggested that if a program is known to consist of $m$ such units, an $m \times m$ Boolean connectivity matrix, $A = [a_{ij}]$, can be constructed where $a_{ij} = 1$ if the $i$th unit (called $\alpha_i$) can transfer control to the $j$th unit (called $\alpha_j$). Otherwise $a_{ij} = 0$. If $a_{ij} = 1$ there is said to be a path of length 1 from $\alpha_i$ to $\alpha_j$. In addition it is possible to construct a row vector $G = (g_1, g_2, \ldots, g_m)$ where $g_i$ is the volume requirement of $\alpha_i$.

## LOOP DETECTION

Figure 2 shows the algorithm used for identifying and merging units forming loops. The basic tool used in the detection of loops is the connectivity matrix, $A$, raised to some power. Lowe has developed a fast, simple technique for multiplying Boolean matrices.[9] If $D = A^n$ and $d_{ij} = 1$, there is a path of length $n$ from $\alpha_i$ to $\alpha_j$. In particular, if $d_{ii} = 1$, there is a loop of path length $n$ involving $\alpha_i$. Furthermore, for all other members, $\alpha_j$, of this loop containing $\alpha_i$, $d_{ij}=1$. However, there may be more than one loop of path length $n$ and therefore there may be more than $n$ values for $i$ such that $d_{ii}=1$. Thus one cannot always uniquely identify the members of a loop by merely inspecting $D$. If $A^n$, $A^{n-1}$, and $A$ are all simultaneously available, this problem is solved.

Let $C=A^{n-1}$ and let $D=A^n$. Choose some $i$ such that $d_{ii}=1$. Then $\alpha_i$ is a member of a loop of path length $n$. If there exists some $j$ such that $c_{ij}=1$, $d_{jj}=1$ and $a_{ji}=1$, then $\alpha_j$ is a member of the same loop. Furthermore, if one were to start at $\alpha_i$ and traverse the loop, the last unit encountered before returning to $\alpha_i$ would be $\alpha_j$. The process could be repeated, looking for $k$ such that $c_{jk}=1$, $d_{kk}=1$, and $a_{kj}=1$, etc., until all $n$ members of the loop have been identified. It should be noted that in addition to identifying these units, this technique also yields information regarding their sequence of execution; i.e., they are detected in reverse order of that in which they would be executed although at this point one cannot tell which will be the initial unit of the loop.*

After a loop has been completely identified, the units comprising the loop are merged in execution order into the first unit detected for this loop. To merge $\alpha_j$ into $\alpha_i$ the following steps must be performed:

1. Replace the $i$th row of $A$ by the logical sum of itself and the $j$th row of $A$.
2. Replace the $i$th column of $A$ by the logical sum of itself and the $j$th column of $A$.
3. Replace $a_{1j}$, $a_{2j}$, . . . $a_{mj}$; $a_{j1}$, $a_{j2}$, . . . $a_{jm}$ and $a_{ii}$ by zero.
4. Replace $g_i$ by $g_i+g_j$.
5. Replace $g_j$ by zero.

It is to be noted that mergers are reflected in $A$ but not in the higher powers of $A$. If, when it is time to merge some member of the loop, say $\alpha_k$, into $\alpha_i$, $g_i$ is such that $g_i+g_k$ exceeds segment size no further mergers are made into $\alpha_i$. Instead the remaining mergers for the loop are made into $\alpha_k$. This reflects the situation where the volume requirement of the loop is such that it exceeds segment size and the loop must be split between two or more pages.

It is clearly not necessary to consider loops with path length of one as no merging of units is necessary in such cases. Therefore if one examines $A^n$ and $A^{n-1}$ for values of $n=2, 3, \ldots, m$ (where $m$ is the dimension

* If it were desired to detect units of a loop in order of execution rather than reverse order one need only use $c_{ji}$ and $a_{ij}$ in place of $c_{ij}$ and $a_{ji}$ respectively.

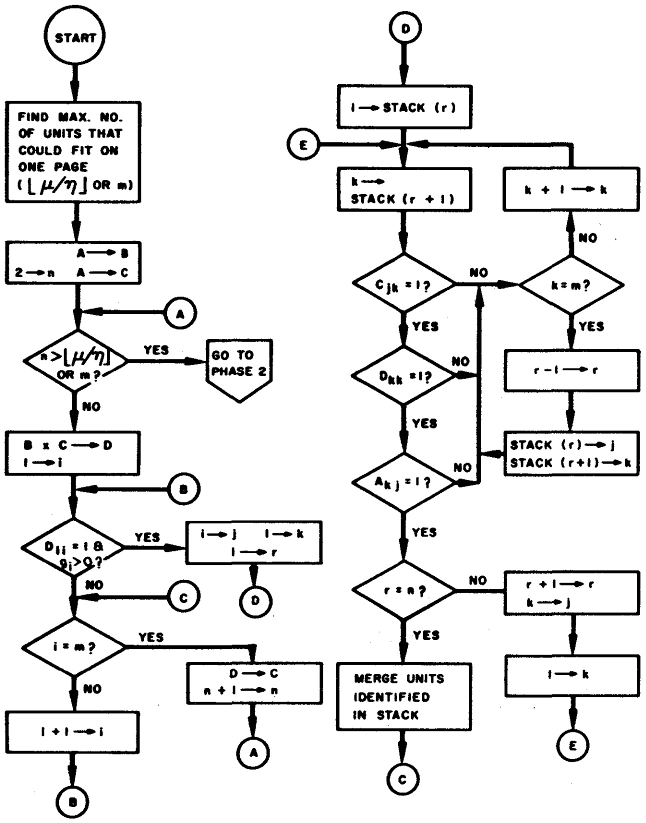

Figure 2—Loop merging

of $A$) one finds successively loops of path length 2, 3, . . . , $m$. If the segment size is $\mu$ and the size of the smallest unit is $\eta$ (both measured in some common unit such as words or instructions), then it is obvious that $\lfloor \mu/\eta \rfloor$* is the largest number of units that could be merged without exceeding segment size. Thus it suffices to limit the above examination to values of $n \leq \min(m, \lfloor \mu/\eta \rfloor)$.

## NON-CYCLIC CONNECTIVITY

As stated earlier, at the start of phase 2 the program consists of units forming a linear string, a tree structure, a lattice structure, loops larger than a segment, or some combination thereof. For illustrative purposes the example shown in Figure 3 suffices.

Figure 4 shows the approach used in phase 2 which is as follows: Find the smallest $i$ such that $g_i>0$. Find the smallest $j$ such that $a_{ij}=1$, $g_j>0$ and $g_i+g_j$ is not greater than segment size. Similarly find the smallest $k$ such that $a_{jk}=1$, $g_k>0$ and $g_i+g_j+g_k$ is not greater than segment size. In this manner one continues to move along some path from $\alpha_i$ to some unit $\alpha_p$ where

* $\lfloor X \rfloor$ is the largest integer less than or equal to $X$

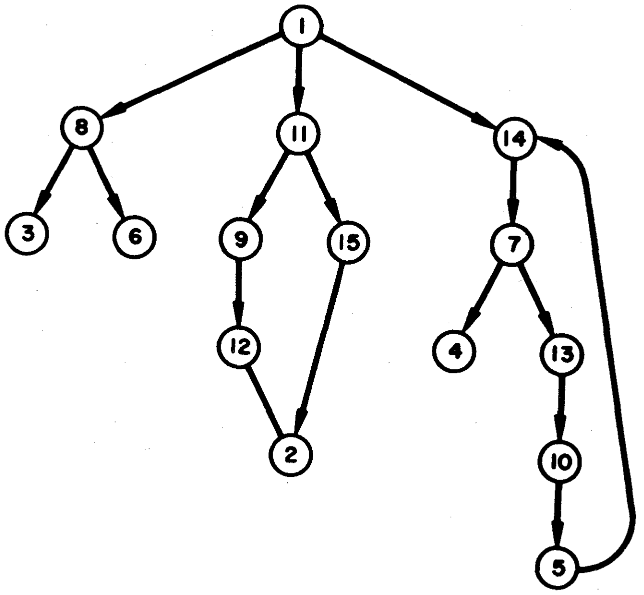

Figure 3—Program structure to be merged

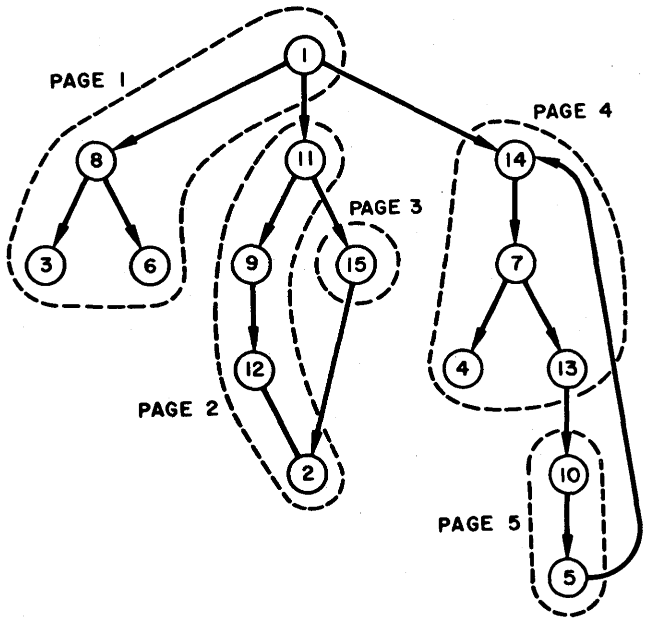

Figure 5—Merged program

either $\alpha_{pq}=0$ for $1 \leq q \leq m$ (i.e., $\alpha_p$ is terminal) or $g_q$, when added to $g_i+g_j+\cdots+g_p$ is greater than segment size. If $\alpha_p$ is not terminal, all units along the path from $\alpha_j$ to $\alpha_p$ are merged into $\alpha_i$, $\alpha_i$ is put into a Last-In, First-Out list, and the path is explored further using $\alpha_q$ as the initial unit. Finally a terminal unit must be encountered. When that happens this unit is merged into the unit which preceded it in this path. If, after the merger, the resultant unit is not terminal, the next branch from that unit is selected as the path to be traversed. If, however, the resultant unit after merger is terminal, it is merged into the unit that preceded it in the path. The procedure will finally result in an initial unit which is terminal. This unit is then marked as having been processed and the last unit in the LIFO list is reselected. A branch from this unit is selected in the above described manner and the resultant path is traversed using the same rules as before. In all the above processes, if ever a branch is selected which leads to a unit that has already been processed (regardless of whether it was actually merged), this fact is recognized and a different branch is selected.

When the LIFO list is finally empty the entire structure emanating from the originally selected unit, $\alpha_i$, will have been merged on a connectivity basis, subject to segment size constraints. However $\alpha_i$ might not be the unit which contains the entry point for the program and the program might even have multiple entry points. Therefore there might be units which were not encountered in the above process. For this reason the units are scanned to determine whether any were not processed. If any such unit is found, the process is repeated starting with that unit. Note however that there will be no repetition of analysis of units that were already processed. Only when all units have been processed will this phase terminate.

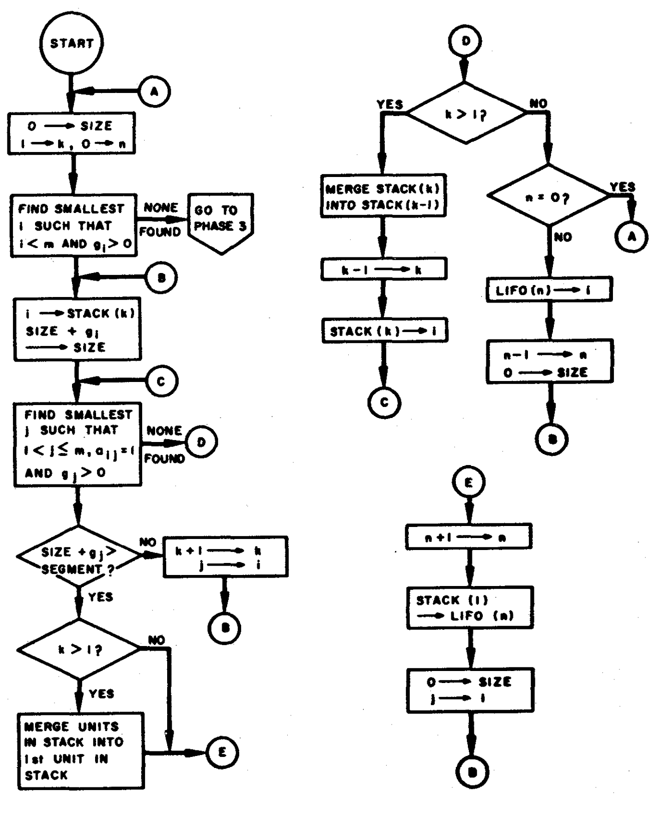

Figure 4—Non-cyclic merging

The result of applying phase 2 to the example shown earlier, assuming each unit was of equal size and segment capacity was four units, is shown in Figure 5.

## CLEANUP

At this point all packing based on connectivity is complete. However, note that there might be units which are less than segment size, such as units 10 and 15. Phase 3 is a clean-up process which attempts to minimize total required storage. In this phase only the unit size vector, $G$, is examined. The algorithm is as follows:

1. Find smallest $i$ such that $g_i > 0$.
2. For $i+1 \leq j \leq m$, if $g_j > 0$ and $g_i + g_j$ is not greater than segment size, merge $\alpha_j$ into $\alpha_i$.
3. Find smallest $i'$ such that $i+1 \leq i' < m$ and $g_i' > 0$.
4. If $i'$ can be found go back to step 2; otherwise the process is complete.

At this point each unit of non-zero size is assigned to some page.

## CONCLUSIONS

The algorithm presented above provides a simple means for partitioning programs into pages in such a manner as to reduce the number of inter-page references and therefore amount of inter-segment activity. It is based solely on information about the program that is obtainable by automatic inspection of the program such as could be preformed in a compiler or assembler. The inspection and packing algorithms are not very complex and thus should require very little time for their execution. The packing algorithm has been implemented as a post-compilation operation. It has been tested with a wide variety of program structure models and has been found to give the anticipated packings. The inspection algorithm is now being incorporated into a JOVIAL compiler specially built for experiments in segmentation.

## REFERENCES

1 R C DALEY P G NEUMAN
*A general-purpose file system for secondary storage*
AFIPS Conference Proceedings Vol 27 Fall Joint Computer Conference Spartan Books Washington DC pp 213-229

2 E L GLASER J F COULEUR G A OLIVER
*System design of a computer for time sharing applications*
AFIPS Conference Proceedings Vol 27 Fall Joint Computer Conference Spartan Books Washington DC pp 197-202

3 V A VYSSOTSKY F J CORBATO R M GRAHAM
*Structure of the MULTICS supervisor*
AFIPS Conference Proceedings Vol 27 Fall Joint Computer Conference Spartan Books Washington DC pp 203-212

4 T C LOWE J G VAN DYKE R A COLILLA
*Program paging and operating algorithms*
RADC Final Report TR-68-444 November 1968
Rome Air Development Center AFSC Griffiss AFB New York

5 C V RAMAMOORTHY
*Analysis of graphs by connectivity considerations*
J ACM 13 2 April 1966 pp 211-222

6 C V RAMAMOORTHY
*The analytic design of a dynamic look-ahead and program segmenting scheme for multi-programmed computers*
Proc ACM 21st Nat Conf 1966 Thompson Book Co Washington DC pp 229-239

7 E W VER HOEF D L SHIRLEY
*Block file and MULTICS systems interface investigation and programming*
Final Report Vol II RADC Final Report TR-69-40 April 1970
Rome Air Development Center AFSC Griffiss AFB New York

8 T C LOWE
*Analysis of boolean models for time-shared paged environments*
Comm ACM 12 4 April 1969 pp 199-205

9 T C LOWE
*An algorithm for rapid calculation of products of boolean matrices*
Software Age 2 March 1968 pp 36-37