Segmentation and optimization of programs from cyclic structure analysis

by JEAN-LOUP BAER and ROBERT CAUGHEY*

University of Washington
Seattle, Washington

INTRODUCTION

Modelling of computer programs by directed graphs, where the vertices represent the computational tasks and the arcs show the flow of control, has been used for optimization purposes,1,10 parallel processing evaluation,7,8 and segmentation.7,9,13 All these studies are mainly based on the fact that a high proportion of the execution time of a program is spent in loops. Although the cyclic structure of programs can be theoretically complex, it has been observed that the nestedness is seldom very deep.6 Thus if one wants to optimize programs written in a high-level language, the detection of cycles by the compiler and its use in an optimization phase may yield a definite improvement in execution time without having to pay too heavily for excess compiling time. In this paper we show how a compiler could make use of an efficient cycle detection algorithm12 to model the embeddedness of cycles by an acyclic directed graph. The latter can then be used for packing purposes in case of a paging or "cache" memory system as well as for predicting the expected number of executions of a given statement for optimization purposes.

MODELLING OF FORTRAN PROGRAMS

For clarity of exposition we present the modelling process as a sequence of four phases. However, actions of phases 1, 2, and some of 3 are performed concurrently. The source program itself is scanned only once. The four phases are now described both functionally and in terms of the basic data structures. Detailed algorithms can be found in Reference 4.

Phase 1

Directed graph model—Instruction Sequences

The basic model is a directed graph $G(W, U)$ where $W$ is the set of vertices $\{w_1, w_2, \ldots, w_n\}$ and $U$ is the set of ordered pairs (or arcs) $u_i = (w_i, w_j)$. Vertices represent computational tasks and arcs show the flow of control. Methods to analyze a FORTRAN source program and transform it into an equivalent—in terms of control—directed graph are now well-known. We shall follow Russell’s approach10 and therefore we assume the uniqueness of an initial vertex $w_1$ and terminal vertex $w_n$ as well as connectedness of the graph. Appendix I shows the vertices and arcs generated from the analysis of the current source statement according to its type. Figure 1 gives an example program (executable statements only) and its directed graph representation. Associated with each vertex $w_i$ is a volume attribute $v_i$ which represents the amount of memory words required for the machine code generated by the translated statement. Thus given a source program, the first part of the modelling process yields $G(W, U)$ which can also be conveniently represented by an $m \times m$ (where $m = |W|$) Boolean connectivity matrix $C_i$ and a volume vector $V(\{V\} = m)$.

For reasons which will become apparent in Phase 3 we now define the Instruction Sequences (I.S.) of a source program.

Let $A$ be the class of all FORTRAN executable source statement types which do not have the attribute of implicit flow of control* and $\bar{A}$ its complement with respect to the class of FORTRAN executable source statements. Let $Z = S_1, S_2, \ldots, S_n$ be the

* Present address: System Services Department, Snohomish County, Everett, Washington.

* I.e., these statements which have as a unique immediate successor the next statement in sequence.
finite string of source statements which comprise a program, in the order (on ascending index) in which they are initially submitted to a translator. An instruction sequence $I_k$ is a non-empty substring of consecutive elements of $Z$ written as

$$I_k = S_{k1}, \ldots, S_{km}$$

such that:

(i) $S_{kj} \in A$ for $1 \leq j \leq m - 1$ and $S_{k1}$ is called the initial statement of $I_k$

(ii) $S_{km} \in A$ and $S_{km}$ is called the terminal statement of $I_k$

(iii) $I_k$ is the substring of $Z$ of maximum length which contains $S_{km}$.

$Z'$ can also be represented by a directed graph $H(I, Y)$ where $I$ is the set of vertices $I_k$ representing the I.S.'s and $y_k = (I_i, I_j)$ is an arc $\in Y$ if the terminal statement $S_{jm}$ of $I_i$ can transfer explicitly to a statement $S_{ip}$ of $I_j$. This directed graph can also be represented by a Boolean connectivity matrix $M$.

Given $Z$ the string of source statements, the determination of the I.S.'s and their interconnections is easy and not time consuming (Figure 2 displays these new structures for the example program of Figure 1). It follows the same algorithms as the one used for the generation of branch instructions by the compiler.

**Phase 2**

**Identify DO-loops and their embeddedness**

In this phase we record the membership of statements in DO-loops in the following way. Each DO-loop is represented by an $m$-dimensional Boolean vector, say $d_i$ for the $i$th DO-loop, such that $d_{ik} = 1$ if the statement represented by vertex $w_k$ is in the DO-loop and $d_{ik} = 0$ otherwise.

1. $I_1 = (1,2)$
2. $I_2 = (3)$
3. $I_3 = (4,5,6,7)$
4. $I_4 = (8,9)$
5. $I_5 = (10,11)$
6. $I_6 = (12)$
7. $I_7 = (13,14,15,16)$
8. $I_8 = (17,18)$
9. $I_9 = (19)$
10. $I_{10} = (20,21,22,23,24)$
11. $I_{11} = (1,2)$
12. $I_{12} = (3)$
13. $I_{13} = (4,5,6,7)$
14. $I_{14} = (8,9)$
15. $I_{15} = (10,11)$
16. $I_{16} = (12)$
17. $I_{17} = (13,14,15,16)$
18. $I_{18} = (17,18)$
19. $I_{19} = (19)$
20. $I_{20} = (20,21,22,23,24)$

(a) Instruction Sequences.

From the collection of the Computer History Museum (www.computerhistory.org)
otherwise. At the same time, we build a (partial) embeddedness Boolean matrix $E^+$ which is such that $e_{ij}^+ = 1$ if DO-loop $j$ is nested within DO-loop $i$.

The construction of the $d_i's$ and $E^+$ is up to this point straightforward and mainly requires the same techniques as those used generally in the compilation of DO-loops, that is the maintenance of a stack of currently active DO-loops and their associated terminating labels. Figure 3 shows the $d_i's$ and $E^+$ after DO-loop processing for the example of Figure 1.

**Phase 3**

*Identify the remaining elementary cycles*

A general algorithm to detect the elementary cycles in a directed graph has been recently published. It is based on a path extension technique with each vertex of the graph being a potential starting point for a cycle. In our case, the searching time of the exhaustive detection—a function of the number of vertices and arcs in the graph—can be greatly reduced by the following two observations.

(i) Ignore the feedback arcs of the DO-loops since the corresponding cycles have already been detected.

(ii) In FORTRAN only a subclass $X$ of statements (XCA) can be origin statements of cycles. This subclass $X$ corresponds to statement types*:

- GO TO $K$
- GO TO $L$, ($K_1, K_2, \ldots, K_p$)
- GO TO ($K_1, K_2, \ldots, K_p$), $L$
- IF (E) $K_1, K_2, K_3$
- IF (B) $S$ where $S$ is a GO TO statement

Only statements of type belonging to $X$ can be starting points of cycles in $G(W, U)$, and they are necessarily terminal statements of I.S.'s. Thus we are going to reduce our cycle search by looking first for cycles in $H(I, Y)$ and then transform these into cycles of $G(W, U)$. This phase is going to be processed in three stages.

**Stage 1. Reduce domain of search of elementary cycles of I.S.'s**

A list of paired origin-destination statements, where the origin statements are the terminal statements of the I.S.'s and the destination statements are those referred to by the labels $K_i$ is built during the scanning of the source program $Z$. The forward reference labelling problem is dealt with in the same manner as the compiler must do for generating code. The corresponding list of origin-destination I.S.'s is also constructed (cf. Table I for our example).

* We restrict ourselves to the analysis of main programs or subroutines, that is we do not include CALL statements as statements which imply transfer of control.

---

**TABLE I—List of Origin-Destination statements and I.S.'s**

<table>
<thead>
<tr>
<th>Origin-Destination</th>
<th>Origin-Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 3</td>
<td>1, 2</td>
</tr>
<tr>
<td>2, 4</td>
<td>1, 3</td>
</tr>
<tr>
<td>2, 20</td>
<td>1, 10</td>
</tr>
<tr>
<td>3, 24</td>
<td>2, 10</td>
</tr>
<tr>
<td>7, 6</td>
<td>3, 3</td>
</tr>
<tr>
<td>7, 8</td>
<td>3, 4</td>
</tr>
<tr>
<td>9, 8</td>
<td>4, 4</td>
</tr>
<tr>
<td>9, 10</td>
<td>4, 5</td>
</tr>
<tr>
<td>11, 10</td>
<td>6, 3</td>
</tr>
<tr>
<td>11, 12</td>
<td>7, 7</td>
</tr>
<tr>
<td>12, 5</td>
<td>8, 3</td>
</tr>
<tr>
<td>16, 15</td>
<td>8, 9</td>
</tr>
<tr>
<td>18, 5</td>
<td>9, 10</td>
</tr>
<tr>
<td>18, 19</td>
<td>9, 2</td>
</tr>
<tr>
<td>19, 20</td>
<td>5, 5</td>
</tr>
<tr>
<td>19, 3</td>
<td>5, 6</td>
</tr>
</tbody>
</table>

---

**Figure 3—Array D (cycle vertex membership) and matrix $E^+$ (cycle embeddedness) after DO-loop processing of the example program**
### Figure 4—Matrix $M^*$ and domain of starter I.S.'s

<table>
<thead>
<tr>
<th>I. S. Number</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The domain of possible starter nodes for the cycle detection algorithm could be further reduced as follows. Let $(I_a, I_b)$ be an origin-destination pair of I.S.'s. They will be part of a same cycle if and only if $m_{ab}^* = m_{ba}^* = 1$ in the reflexive transitive closure $M^*$ of the connectivity matrix $M$. If these relations do not hold the origin-destination pair can be deleted. However, it is debatable if the gain incurred by this deletion surpasses the cost of generating $M^*$, a process of order greater than $n^2$.

### Stage 2. Find elementary cycles of I.S.'s

The algorithm already referred to is used with the short-cut of skipping those vertices which are not part of the reduced set of possible starting points. Figure 5 lists these cycles.

### Stage 3. Conversion of I.S. cycles to original graph cycles (G-cycles)

Each cycle of I.S.'s must be converted into cycles of the original graph (G-cycles). It is of course possible that a single I.S. cycle may yield several G-cycles. For example, the I.S.:

<table>
<thead>
<tr>
<th>Corresponding vertex number</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 A=B</td>
</tr>
<tr>
<td>20 C=D</td>
</tr>
<tr>
<td>IF (E)10,20,30</td>
</tr>
</tbody>
</table>

which has one I.S. cycle, namely itself, yields two G-cycles [1, 2, 3] and [2, 3].

The conversion algorithm may be summarized as follows: Let $C_i = I_a, I_b, \ldots, I_\sigma$ be an I.S. cycle. Recalling the definition of an I.S. one can write $I_a = S_{a,1}, S_{a,2}, \ldots, S_{a,ma}$. All $S_{a,ma}$ are part of all G-cycles generated by $C_i$. Let $(S_{a,ma}, S_{b,1})$ be a paired origin-destination statement leading from $I_a$ to $I_b$. Now the sets $d_i$ corresponding to G-cycles are obtained as follows:

1. Initialization step. $d_\phi = \phi$ (the empty set).
2. Let $\alpha$ be the index of the first I.S. in $C_i$ and $\beta$ the index of the next I.S. in sequence (if there is only one I.S. $\beta = \alpha$).

$$d_\alpha = d_\beta \cup \{S_{a,ma}\}$$

### Figure 5—List of cycles of instruction sequences

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>(3)</td>
</tr>
<tr>
<td>$C_2$</td>
<td>(3, 4, 5, 6)</td>
</tr>
<tr>
<td>$C_3$</td>
<td>(3, 4, 5, 6, 7, 8)</td>
</tr>
<tr>
<td>$C_4$</td>
<td>(3)</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
3. For each origin-destination pair \((S_{a,m}, S_{b,j})\) do:
   For each \(d_i\) do:
     \[d_i^D = d_i \cup \{S_{b,j}, \ldots, S_{a,m}\}\]

4. \(\alpha = \beta = \gamma\) (set to index of next I.S.). (If \(\alpha = \sigma\),
   \(\beta\) is set to the index of the first I.S.)

If \(\alpha = \) index of the first I.S. then stop; the \(d_i\)'s are the
last \(d_i\)\(\beta\) generated; else go to step 3. Figure 6 shows the
set of \(G\)-cycles generated in the Boolean notation
adopted in phase 2.

**Phase 4**

**Complete and reduce the embeddedness matrix \(E^+\)**

We now want to complete \(E^+\) which was defined and
partially built during phase 2. If \(d_i\) and \(d_j\) are two
\(G\)-cycles in Boolean representation, then let \(f_{ij} = d_i \cap d_j\).
Elements of \(E^+\) will now be defined by:

- If \(f_{ij} = d_i\) then \(e_{ii}^+ = 1, e_{ii}^+ = 0\) (\(d_i\) embedded
  in \(d_j\))
- If \(f_{ij} = d_j\) then \(e_{ij}^+ = 0, e_{jj}^+ = 1\) (\(d_j\) embedded
  in \(d_i\))
- If \(f_{ij} \neq d_i \neq d_j\) then \(e_{ij}^+ = e_{ji}^+ = 0\).

Since the embeddedness of the DO-loops between
themselves have already been recorded, these tests are
done only for the pairs of cycles \(i, j\) where \(1 \leq i \leq n\)
\((n\) number of cycles\), \(k < j \leq n\) (\(k\) number of DO-loops).

Now the \(E^+\) matrix can be considered as the precedence
matrix of an acyclic directed graph where the
embeddedness property is the precedence relation, i.e.,
the vertex representing \(d_i\) is a successor of the one
represented by \(d_j\) if \(e_{ji}^+ = 1\). In the sequel we shall use
strict embeddedness which is defined as: \(d_i\) is strictly
embedded in \(d_j\) if it is embedded in \(d_j\) and there exists
no \(d_k\) such that \(d_i\) is embedded in \(d_k\) and \(d_k\) is embedded
in \(d_j\). In order to find a representation of strict embeddedness one has to find the minimum connectivity
matrix \(E\) such that \(E^+ = \bigcup_{i=1}^n E^i\). Solutions to this
problem can be found in Reference 3, 11. Figure 7 shows \(E\)
for our example program.

This construction completes our modelling process.
We now describe some possible applications.
### TABLE II—Matrix S

<table>
<thead>
<tr>
<th>I</th>
<th>MAXLEVEL</th>
<th>USED</th>
<th>NAME</th>
<th>LEVEL</th>
<th>DEGREE</th>
<th>FATHER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### APPLICATION OF THE MODEL TO PAGING AND CACHE MEMORY SYSTEMS

The model can be used as an aid in packing code in pages in such a way that the number of interpage transitions due to instructions is reduced in comparison with the compiler's usual memory allocation technique. Other attempts at solving this problem range from a macroscopic approach to the analysis of generated code. We will follow a middle of the road route similar in its degree of detail to other studies. We do not try in any way to achieve an optimal packing such as in Reference 14 which not only requires information that the model cannot deliver such as the cost of page faulting but which would also take too much computing at compilation time. Our goal is to reduce the number of page transitions through easily implementable algorithms with the understanding that some gross assumptions may have to be taken. The basic philosophy is to pack first those statements which are in the most nested cycles. This was the purpose of obtaining the matrix $E$.

We need a few additional definitions. Extending the usual relations found in trees, we define the roots of an acyclic directed graph to be those vertices which have no predecessors. The immediate successors of a vertex will be its sons, its immediate predecessors will be its fathers, and sons of the same father will be brothers. Note that this last relation is transitive only in the case of a tree. The level $l_i$ of a vertex $w_i$ is the length of the longest path from a root to $w_i$. The maximum level $m_l$ is defined as $\max(l_i, l_j, \ldots, l_k)$ where $w_j, \ldots, w_k$ are the successors of $w_i$. These levels $l_i$ and $m_l$ can be computed directly from $E^+$ or by performing a topological sort on the graph.

#### Traversal of the graph

Our first objective is to traverse the graph in an order reflective of the nestedness of cycles. A sort on levels $l_i$, which satisfies this criterion, is rejected because at the same time we want to keep as much as possible the image of the structure of the source program. To facilitate the traversal, the graph is going to be modified into a forest. If in the original graph the vertex $w_i$ had more than one father, the one selected to stay is the one with greatest maximum level. That is for all columns $i$ which have more than one bit set to 1 in matrix $E$, let $j$ be the index such that $e_{ji} = e_{ki} = e_{li} = \cdots = e_{pi} = 1$ and $m_{lj} = \max(m_{lj}, m_{ki}, m_{li}, \ldots, m_{pi})$. Then set all bits of column $i$ to 0, except for $e_{ji}$ which is set to 1. After such a transformation is completed the levels and maximum levels have to be recomputed. The forest can now be represented in a tabular form (cf. Table II) which aids in the traversal of the forest. This traversal, a variation on post-order is defined recursively by:

1. Order the sons of the root $r$ by descending maximum levels in $\text{SONS}_r$, say $(r_1, r_2, \ldots, r_k)$.
2. Traverse the sub trees of roots $r_1, r_2, \ldots, r_k$ in that order. When a subtree is composed only of a single vertex, its root, the root is then traversed.

In the tabular form, the field USED is a flag showing those vertices which have already been traversed, DEGREE is a counter for the sons which have not yet been traversed and the other fields are self-explanatory. For our example, the algorithm yields $\{914, 7856\}$ and $\{32\}$ for the traversal of the two trees.

During the traversal of the forest, the volume of each individual cycle is computed as follows: Let $d_i$ be a cycle, $v_{d_i}$ its volume, $(w_{a1}, w_{a2}, \ldots, w_{a\theta})$ its members and $(v_{a1}, v_{a2}, \ldots, v_{a\theta})$ be the volumes of the $w_{ai}$. Set $d$ is the set of vertices which have yet to be traversed (initially $d=W$) and $d_i$ is the cycle currently traversed. Then if $d' = d \cap d_i$, $v_{d_i} = \sum_{w_{a1} \in d_i} v_{a1}$ and one suppresses from $d$ all vertices belonging to $d_i$. 

From the collection of the Computer History Museum (www.computerhistory.org)
Assign vertices to pages

Given i) a page size \( p \), ii) the order in which cycles are traversed per the above algorithm, say \( T = \{ d_{i1}, d_{i2}, \ldots, d_{ik} \} \) and iii) their volumes \( VD = \{ vd_{i1}, vd_{i2}, \ldots, vd_{ik} \} \) an algorithm to assign vertices to pages, a direct descendant of Lowe's, is summarized here. The algorithm accesses the cycles in \( T \) consecutively left to right, assigning as many cycles \( d_{ij} \) (each cycle in its entirety) to a single page as the memory volumes \( vd_{ij} \) permit. By a cycle in its entirety is meant that a cycle in its vertex assignment is not permitted to run off the end of one page and have following vertices assigned to a subsequent page except when the cycle is the initial (and hence sole) cycle assigned to the page. Whether the cycle is assigned to a page as in the page over-run case or in aggregate vertex by vertex as in the former case, the assignment is reflected by a merger of graph vertices in connectivity matrix \( C \) and by the merger of memory volumes in the volume vector \( V \) (cf. phase 2 of the second section).

The algorithm consists of three parts. The first part deals with initialization for assigning a new cycle from \( T \) to pages. The second considers assignments to pages when a cycle is known to fit entirely within a page and is therefore not assigned to the page on a vertex by vertex basis. The third part considers the case when a cycle exceeds the length of a page and has its vertices assigned to the initial and following pages on a vertex by vertex basis. The algorithm terminates when the cycles in \( T \) have been exhausted.

A step by step description of the algorithm follows.

0. Set the current page size \( cp \) to 0. Set \( h = 0 \).

A. [Initialization].
1. If \( h = k \) (i.e., vector \( T \) completely traversed) then end.
2. Let \( h \) be the index of the next (initially first) cycle in vector \( T \) and \( d_j \) be the Boolean representation of this cycle.
   \[ h = h + 1; \quad j = T(h) \]
3. Let \( n \) be an index varying from 1 to \( m \) (the number of vertices in \( G \)). Set \( n = 0 \).
4. If \( cp = 0 \) go to step 7.
5. If \( cp + vd_j \leq p \) (i.e., if the current cycle will not go over a page boundary) go to step 11.
6. Set \( cp = 0 \) (Start a new page).
7. \( n = n + 1 \).
8. If \( d_m = 0 \) (i.e., vertex \( n \) does not belong to cycle \( j \)) go to step 7.

B. [The cycle fits in (remainder of) page].
9. \( q = n \) (\( q \) temporary variable for last vertex traversed).
10. If \( vd_q > p \) (if cycle does not fit entirely in a page) go to step 16 (part C).
11. \( cp = cp + vd_j; \quad v_q = cp \) (This is part of the merging process).
12. \( n = n + 1 \).
13. If \( d_m = 0 \) go to step 15.
14. \( v_n = 0 \). Merge \( w_n \) into \( w_q \) (for an explanation of merging see below).
15. If \( n = m \) go to step 1 (i.e., to part A) else go to step 12.

C. [The cycle overflows a page].
16. \( cp = v_n \).
17. \( n = n + 1 \).
18. If \( d_m = 0 \) go to step 22.
19. If \( v_q + v_n > p \) go to step 21.
20. Merge \( w_n \) into \( w_q \) and \( v_n \) into \( v_q \); \( cp = cp + v_n \); Go to step 22.
21. \( q = n; \quad cp = v_n \).
22. If \( n = m \) go to step 1 else go to step 17.
TABLE III—Volume Vector for Example Program

<table>
<thead>
<tr>
<th>Vertex</th>
<th>1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>8 4 2 2 3 3 3 3 3 3 3 5 5 6 5 4 3 4 4 5 5 8 3 2</td>
</tr>
</tbody>
</table>

The merging process is defined by:

- To merge $w_i$ into $w_q$, replace the $q$th row of $C$ by the logical sum of itself and the $n$th row of $C$, and replace the $q$th column of $C$ by the logical sum of itself and the $n$th column of $C$. Set the $n$th column and row of $C$ to all 0's.

- To merge $v_n$ into $v_q$, replace $v_q$ by $v_q + v_n$ in the memory volume vector $V$ and set $v_n$ to 0.

The remaining task is to assign those nodes which are not members of cycles. It is done by following part C above, after first initializing both $q$ and $n$ to 1.

Figure 8 shows the result of packing for $p = 8$ (a very small page size only for illustration purpose) and the volume vector of Table III. Although this algorithm is crude—and we see below how to improve it—it is worthwhile to compare its outcome with what would have been obtained by usual compiling techniques. Figure 9 shows how the code would be allocated in a straightforward manner with vertices allowed to run over pages.

As can be seen the total number of pages required for the generated code is smaller in the usual compiler's case (7 instead of 10). However, the goal of the packing algorithm is to improve the residency of the pages, i.e., the locality of the code. In general the number of data pages needed for efficient execution is much greater than the number of code pages; the packing algorithm thus helps in leaving more main memory space to data during long bursts of execution. For example, loops $(w_6, w_7)$, $(w_{10}, w_{11})$, and $(w_{22})$ occupy one page instead of two. Moreover, as stated earlier, the packing algorithm can be improved. A first clean-up pass could be implemented as follows:

If after the merging process, $C_{ij} = 1$ (i.e., pages $i$ and $j$ are connected) and $v_i + v_j < p$ (the sum of their memory volumes is less than a page), then merge $w_i$ into $w_j$ and $v_i$ into $v_j$.

In our example, pages 16 and 17 would be merged, as well as pages 18 and 19, thus leading to a definite improvement in the second part of the main cycle.

A more sophisticated clean-up pass would be first to "squeeze" those elementary cycles which are leaves of the forest, by allowing vertices to run over pages. (In our example $w_{24}$ would be in pages 15 and 16.) Then the clean-up pass defined above would be invoked. Figure 10 shows the results of the improved algorithm. The main loop requires now only 8 pages; the number of interpage transitions is diminished, and the locality of the code is improved.

EXPECTED NUMBER OF EXECUTIONS OF SOURCE STATEMENTS

The knowledge of the expected number of times a given source statement is going to be executed can be an important factor in the optimization of high-level source programs. It allows replacement of compiler-generated code by a tighter hand-coded version in the
event of an often-repeated statement. In the context of the previous section, the traversal of the tree could be monitored by the maximum expected number of executions, instead of being directed by maximum levels, thus leading to a better residency of pages of code.

In this section we show how these numbers can be estimated when i) the probabilities \( q_{ij} \) of traversing the arcs \((w_i, w_j)\) corresponding to paired origin-destination statements are known, and ii) when the mean number of times the feedback arc of a DO-loop is set up to be traversed is known. These parameters of the source program will have to be "guessed" or instrumented. In the former case they represent a first pass at the analysis of the program and only well-known sections of it should be optimized. In the latter, they correspond to an "a posteriori" optimization of a production run. In both cases, we assume that we have a Markovian process, i.e., the \( q_{ij} \) stay invariant each time \( w_i \) is reached.

Traversals of the directed graph

The method we outline is an outgrowth of studies related to the prediction of performance of multiprocessors. Let \( f_i \) be the expected number of executions of vertex \( w_i \), \( p_i \) its probability of being reached from the initial vertex \( w_i \) when all feedback arcs of the graph have been removed. In the case of a stochastic cycle, i.e., one which is not a DO-loop, this feedback arc is the one corresponding to the last paired origin-destination statement encountered when transforming the I.S. cycles to G-cycles, and of probability not equal to 1. After normalizing the probabilities of arcs incident out of vertices which are origins of feedback arcs, one computes the \( p_i \) as:

\[
(1) \quad p_i = \sum_{j=1}^{k} p_j q_{ij} \quad \text{where} \quad w_j \quad (j = 1, \ldots, k) \quad \text{are the immediate predecessors of} \quad w_i. \quad \text{At initialization all} \quad f_i \quad \text{are set to 1.}
\]

The traversal of the directed graph represented by the original embeddedness matrix \( E \) is performed as follows.

1. Let \( d_1, d_2, \ldots, d_k \) be the leaves of the graph. Partition these leaves into classes \( E_1 = \{d_1\}, E_2 = \{d_2\}, \ldots, E_k = \{d_k\} \).
2. If two classes \( E_i \) and \( E_j \) are such that there exists \( d_i \in E_i \) and \( d_j \in E_j \) such that \( d_i \cap d_j \neq \emptyset \), let \( E_i = E_i \cup E_j \) and suppress \( E_j \).
3. Repeat step 2 until no suppression occurs.
4. Compute the \( f_i \) (up to that point) of each vertex member of a cycle belonging to one of the \( E_i \) left after step 3, as explained below.
5. Suppress from \( E \) all the leaves and repeat steps 1 to 4 until all cycles have been treated.

Steps 1 to 3 are facilitated by the use of the tabular representation of the graph. They involve mainly the partition of cycles into classes with the equivalence relation being "to have a common father with at least one member of the class."

Computation of \( f_i \) 's corresponding to leaf cycles

Let \( E_k = \{d_{o1}, d_{o2}, \ldots, d_{ok}\} \) be a class as defined above. We shall give expressions for the \( f_i \) of vertex members of \( d_{oi} \) in supposing first that the cycles are at their maximum depth of embeddedness.
Figure 11—Regular Markov chain (subcase 1 of case 1)

Case 1. \( E_k \) has a unique member \( d_n = \{w_h, \ldots, w_i, \ldots, w_t\} \) where \((w_t, w_h)\) is the feedback arc.

Let \( p_i \) be computed as in (1) in considering only the subgraph corresponding to \( d_n \) and \( p_h = 1 \). We note \( \Pr[w_i(j)] \) as the probability of executing \( w_i \) and \( j \) times only.

Subcase 1. There is no branching out of \( d_n \) that is there exists no arc \((w_i, w_i')\) such that \( w_i \in d_n - \{w_i\} \) and \( w_i' \notin d_n \).

Let \( (N-1) \) be the mean number of times the DO-loop has been set up for and \( p \) the probability of the feedback arc of a stochastic cycle. For a DO-loop:

\[
f = \sum_{j=1}^{N} j \Pr[w_i(j)]
\]

and

\[
f_h = N.
\]

But

\[
\Pr[w_i(j)] = \binom{N}{j} p_i^j (1 - p_i)^{N-j}.
\]

Hence:

\[
f = \sum_{j=1}^{N} \binom{N}{j} p_i^j (1 - p_i)^{N-j}
\]

\[
= N p_i \sum_{j=1}^{N} \binom{N-1}{j-1} p_i (1 - p_i)^{(N-1)-(j-1)}
\]

or

\[
f = N p_i f_h = f_h p_i.
\]

For a stochastic cycle: the process can be considered as a regular Markov chain as shown in Figure 11.a. by introducing an extra arc of probability 1. Now if \( P \) is the regular transition matrix, \((\Pi_1, \ldots, \Pi_n, \ldots, \Pi_n, \Pi_n)\) the limiting probability vector, it is known that the mean first passage time to return to a given state is \( m_{ii} = 1/\Pi_i \). In our case, \( m_{ee} = 1/\Pi_e \) represents the mean number of steps, and \( f_i = \Pi_i/\Pi_e \).

If one is interested only in \( f_i, f_h, f_h \), the graph is equivalent to the one in Figure 11.b. whose transition matrix is:

\[
\begin{array}{cccccc}
0 & p_i & 1-p_i & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 \\
p & 0 & 0 & 0 & 1-p \\
1 & 0 & 0 & 0 & 0
\end{array}
\]

and since \( \Pi_i = p_i \Pi_e \), then \( f_i = p_i f_h \). Now \( f_h \) could be computed by the ratio \( \Pi_h/\Pi_e \) but directly one can see that:

\[
f_h = \sum_{j=1}^{N} j \Pr[w_h(j)] = \sum_{j=1}^{N} j p^{j-1} (1-p) = \frac{1}{1-p}
\]

Subcase 2. There is some branching out of total
probability 1−Q where Q = p_i^*. In the DO-loop case:

\[ P_r[w_i(j)] = Q^{j-1}(1-Q) \quad (j = 1, \ldots, N-1); \]
\[ P_r[w_i(N)] = Q^N. \]

Hence

\[ f_k = \sum_{j=1}^{N} jQ^{j-1}(1-Q) + NQ^{N-1} = \frac{1-Q^N}{1-Q}. \]

Now

\[ P_r[w_i(k)] = \sum_{j=1}^{N} P_r[w_i(j)] \left( \begin{array}{c} j \\ k \end{array} \right) p_i^{s_i}(1-p_i^*)^{j-k} \]

\[ (k = 1, \ldots, N) \]

and

\[ f_i = \sum_{k=1}^{N} \sum_{k=1}^{N} k P_r[w_i(k)] \]

\[ = \sum_{k=1}^{N} \sum_{j=1}^{N} \left( \begin{array}{c} j \\ k \end{array} \right) p_i^{s_i}(1-p_i^*)^{j-k} \]

\[ = p_i^* \sum_{k=1}^{N} k P_r[w_i(k)]. \]

In the stochastic case, the computation of $f_k$ corresponds to the Markov chain of Figure 12 of transition matrix

\[ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & Q & 1-Q \\ \rho & 0 & 0 & 1-\rho \\ 1 & 0 & 0 & 0 \end{bmatrix} \]

which yields

\[ f_k = \frac{1}{1-Q\rho} \]

Analysis similar to subcase 1 would yield $f_i = p_i^*f_k$.

In summary for this first case, the computation of the $f_i$ involve only the computation of the $p_i^*$ and of the $f_k$ as given above.

**Case 2.** $E_k$ has more than one member.

**Subcase 1.** All cycles are stochastic.

Again we have recourse to the Markov chain approach. Of interest are the $f_i$ of the heads and tails of the cycles. A typical example is shown in Figure 12 of transition matrix;

\[ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \rho & 0 & 0 & 1-\rho \\ \rho' & 0 & 0 & 1-\rho' \\ 1 & 0 & 0 & 0 \end{bmatrix} \]
Subcase 2. One member of $E_k$ is a DO-loop (there can be no more than one due to the Syntax of FORTRAN).

Consider again Figure 13 but suppose now that the second cycle is a DO-loop. It could be possible by using methods of renewal theory to determine the $f_i$ in this case. However, we can still use the Markov chain approach. To do so, an $N$-folding of the structure must be performed as shown in Figure 14a. To obtain $f_{h1}$ and $f_{a1}$, a reduction as in Figure 14b is performed, giving a transition matrix

$$
\begin{pmatrix}
\rho & 1-\rho & 0 & \cdots & 0 \\
\rho & 0 & 1-\rho & \cdots & 0 \\
P & \cdots & \cdots & \cdots & P \\
1 & 0 & 0 & \cdots & 0
\end{pmatrix}
$$

which yields

$$
f_{h1} = \frac{1}{(1-\rho)^N},
$$

$$
f_{a1} = \sum_{i=1}^{N} f_{h1} = \sum_{i=1}^{N} \frac{1}{(1-\rho)^i} = \frac{1 - (1-\rho)^N}{\rho(1-\rho)^N}
$$

and $f_{h2}, f_{a2} = (1-\rho)f_{h1}$ as in subcase 1.

If for both cases 1 and 2, the cycles are not embedded in any other cycle, the $f_i$ are multiplied by $p_h$, the probability of reaching the first head. If they are embedded in other cycles, the $f_i$ computed at the next level will be multiplied by those just obtained until all cycles have been treated.

CONCLUSION

In this paper, we have presented a model of Fortran source programs with particular emphasis on the cyclic structure. Applications of the model to segmentation and optimization have been introduced. Algorithms needed to extract information from the high-level language in order to build the model have been given. A packing algorithm, whose goal is to reduce the number of interpage transitions in a paging or cache memory system, has been presented along with its application to an example program. The cyclic structure of the program has also been used to compute the expected number of executions of each statement of the source program.

The results of the packing algorithm, which is meant to increase the locality of the generated code, could also be used in paging mechanisms with anticipatory control. The expected numbers of executions could serve in the packing of data into pages, the data with high correlation being allocated in the same pages. This latter problem needs more investigation and it is in this
direction that extensions of the model and new algorithms should prove useful.

REFERENCES

1 F E ALLEN
Program optimization
Ann Rev in Aut Prog 5: pp 239-302 1969
2 J L BAER
Graph models of computations in computer systems
PhD Dissert UCLA 1968
3 J L BAER
Matrice de connexion minimale d’une matrice de précédence donnée
RIRO 16: pp 65-73 1969
4 R T CAUGHEY
Automatic segmentation of FORTRAN programs from cyclic structure analysis
MS Thesis Univ of Washington 1971
5 F H DEARNLEY G B NEWELL
Automatic segmentation of programs for a two-level store computer
The Computer Journal 7 3 pp 185-187 May 1968
6 D E KNUTH
An empirical study of FORTRAN programs
7 T C LOWE
Automatic segmentation of cyclic structures based on connectivity and processor timing
Comm ACM 13 1 pp 3-9 Jan 1970
8 D F MARTIN G ESTRIN
Models of computational systems cyclic to acyclic graph transformations
Trans IEEE EC-16 pp 70-79 Feb 1967
9 C V RAMAMOORTHY
The analytical design of a dynamic look-ahead and program segmenting scheme for multiprogrammed computers
Proc ACM 21st Nat Conf Thompson Book Co pp 229-239 1966
10 E C RUSSELL G ESTRIN
Measurement based automatic analysis of FORTRAN programs
Spring Joint Conf Proc 34 pp 723-732 1969
11 J M S SIMOES-PEREIRA
On the Boolean Matrix Equation $M' = \bigcup M_i$
Journal ACM 12 3 pp 376-382 July 1965
12 J C TIERNAN
An efficient search algorithm to find the elementary circuits of a graph
13 E W VERHOEF
Automatic program segmentation based on Boolean connectivity
Spring Joint Comp Conf Proc 38 pp 491-495 1971
14 B W KERNIGHAM
Optimal sequential partitions of graphs
Journal ACM 18 1 pp 34-40 Jan 1971
15 J KRAL
One way of estimating frequencies of jumps in programs
Comm ACM 11 7 pp 475-479 July 1968
APPENDIX I

Vertices and arcs generated by the graph modeling process.
Notation: \( w_i \) current vertex number; \( w_{i+1} \) next (sequential) vertex number; \( w_n \) vertex corresponding to source statement of label \( n \); \( w_z \) terminal vertex.

<table>
<thead>
<tr>
<th>STATEMENT</th>
<th>VERTEX GENERATED</th>
<th>ARCS GENERATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO TO ( n )</td>
<td>( w_i )</td>
<td>( (w_i,w_a) )</td>
</tr>
<tr>
<td>GO TO ( (n_1,n_2,\ldots,n_k) ), ( j )</td>
<td>( w_i )</td>
<td>( (w_i,w_{a_1}),\ldots,(w_i,w_{a_k}) )</td>
</tr>
<tr>
<td>GO TO ( j; (n_1,n_2,\ldots,n_k) )</td>
<td>( w_i )</td>
<td>( (w_i,w_{a_1}),\ldots,(w_i,w_{a_k}) )</td>
</tr>
<tr>
<td>IF ( (e) ) ( n_1,n_2,n_3 )</td>
<td>( w_i )</td>
<td>( (w_i,w_{a_1}),\ldots,(w_i,w_{a_k}) )</td>
</tr>
<tr>
<td>IF ( (e) ) ( S )</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>(a) If ( S ) is not a GO TO RETURN or STOP</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>(b) If ( S ) is a GO TO RETURN or STOP</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}),\ldots,(w_i,w_{i+k}) )</td>
</tr>
<tr>
<td>DO ( n ) ( I = M_1,M_2,M_3 )</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}),\ldots,(w_i,w_{i+k}) )</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>PAUSE</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>RETURN</td>
<td>( w_i )</td>
<td>( (w_i,w_a) )</td>
</tr>
<tr>
<td>STOP</td>
<td>( w_i )</td>
<td>( (w_i,w_a) )</td>
</tr>
<tr>
<td>READ( (a,b,END=n_1,\text{ERR}=n_2) ) ( \text{list} )</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}),\ldots,(w_i,w_{i+k}) )</td>
</tr>
<tr>
<td>WRITE( (a,b) ) ( \text{list} )</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>ENDFILE</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>REWIND</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>REWIND</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>Arith. Stats.</td>
<td>( w_i )</td>
<td>( (w_i,w_{i+1}) )</td>
</tr>
<tr>
<td>END</td>
<td>( w_z )</td>
<td></td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)