An automatic overlay generator

by Ron Cytron
Paul G. Loewner

We present an algorithm for automatically generating an overlay structure for a program, with the goal of reducing the primary storage requirements of that program. Subject to the constraints of intermodule dependences, the algorithm can either find a maximal overlay structure or find an overlay structure that, where possible, restricts the program to a specified amount of primary storage. Results are presented from applying this algorithm to three substantial programs.

1. Introduction

In response to the storage demands of complex programming projects, the trend in language support has been to offer programmers increasingly sophisticated address spaces. The references to locations within these *logical address spaces* are specified by a programmer, either directly through machine language or indirectly through a language processor. The logical address space is an abstraction of a machine's *physical address space* in several ways:

1. The logical address space may be of a different size than the physical address space.
2. The logical address space may appear contiguous, but the physical address space may be discontiguous.
3. References to the logical address space may appear identical in cost, but references to the physical address space may vary in cost.
4. The names by which a program references the logical address space may differ from the names offered by the physical address space.

In each case, the logical address space simplifies a programmer's view of storage by hiding certain implementation details of the physical address space. Automatic mechanisms, in the form of *paging* and/or *segmentation*, translate references in the logical address space to locations in the physical address space.

This paper is concerned with one abstraction offered by a logical address space: A program can reference more locations than exist in the physical address space. In particular, we wish to consider an alternative to accommodating this abstraction by paging or segmentation. We propose to automate the heretofore manual process of *overlaying*, by which a logical address space L_1 is mapped into another, potentially smaller, logical address space L_2. One consideration in favor of overlaying is the observation that low-end machines may lack the hardware to accomplish paging or segmentation. Overlaying can allow large programs, previously restricted to mainframe computers, to execute on personal computers.

Overlaying occurs prior to the introduction of a physical address space, as shown in **Figure 1**. A program P consists of a set of M modules $\{m_1, m_2, \cdots, m_M\}$ and each module m_i offers a set of entry points E_i. Because entry points must be known to all modules, they exist at a global level, so $\cap E_i = \varnothing$.

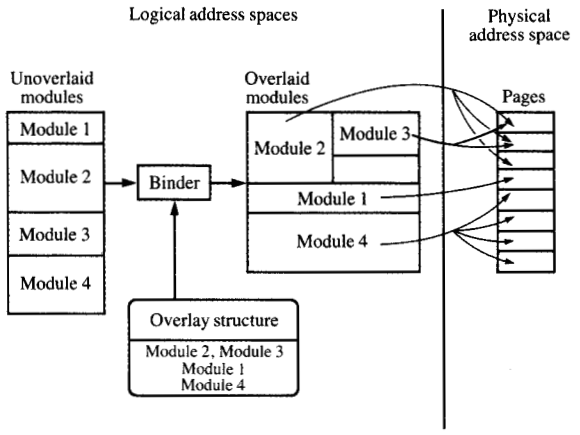

Figure 1

Logical and physical address spaces.

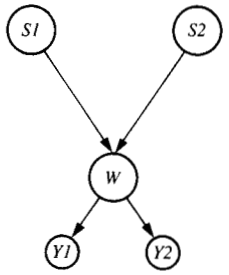

Figure 2

A problem case.

Two modules of the logical address space L_1 are overlaid in L_2 if they share portions of the logical address space L_2.

In the manual overlaying process, a user constructs an overlay structure that specifies those modules that can share portions of the logical address space. This overlay structure is given to a *binder*, or *linkage editor*, that normally resolves references among the modules of a program: Code within some module m_i could reference an entry point within module m_j, and the binder resolves such references in the logical address space L_2. In the absence of an overlay structure, a binder typically assigns logical addresses in increasing order, placing the modules in disjoint portions of L_2. With an overlay structure, the binder is directed to start multiple modules at the same logical address in L_2.

There are several advantages associated with automatic overlaying:

1. Modification of a program can alter intermodule relationships, requiring a concomitant modification of the overlay structure.
2. Failure to understand the relationships among the entry points of a program's modules can result in an *unsafe* overlay structure. An unsafe structure allows two modules m_i and m_k to share portions of L_2 when the residency of both modules is required. (For example, m_i calls m_k and m_k calls m_i.)
3. Maximum reduction of logical address space can require a complex and quite lengthy overlay structure. Even those users who understand intermodule relationships can introduce errors into the specification of the overlay structure.

In consideration of the above points, we propose an automatic method for generating an overlay structure. In the following sections, we discuss our method and show its application to several large programs. Our algorithm produces a maximal overlay structure for a program, in the sense of minimizing the size of the logical address space.

2. Method

In this section, we present an algorithm that derives an overlay structure for a program through the following steps:

1. Generate the call graph.
2. Collapse each strongly connected component of the call graph to create a directed acyclic graph (DAG).
3. Generate an overlay structure from the DAG.
4. Output the overlay structure in a form acceptable by a binder.

Generation of the call graph

The call graph is produced using a program that accepts a set of M modules in their compiled form. Each module m_i offers a set of entry points E_i ($\{e_{ij}\}$, $1 \le j \le |E_i|$), and references a set of entry points R_i ($\{r_{ij}\}$, $1 \le j \le |R_i|$). The program constructs a call graph, with nodes corresponding to the modules of a program. An arc exists between nodes m_i and m_j if m_i references an entry point declared in module m_j ($R_i \cap E_j \neq \emptyset$.)

Intuitively, the relevance of the call graph to the automatic overlay problem is apparent through the following observation: The absence of an arc between modules m_i and m_j implies that m_i cannot directly invoke m_j. Let a *call chain* be defined as a path through the call graph. If no path exists between two modules, then the two modules cannot be active simultaneously, since there is no call chain that can cause m_j to be invoked until m_i has returned. It is precisely such circumstances that allow m_i and m_j to be overlaid.

• *Collapsing strongly connected components*

This step derives a DAG from the call graph by collapsing the strongly connected components of a call graph into single nodes using a method due to Tarjan [1]. Thus, any modules that can execute recursively are represented by a single node. This step simplifies subsequent transformations, with no loss of generality: Modules that participate in recursive invocations require the residency of each other, prohibiting them from sharing portions of logical address space. The DAG is created from the call graph as follows:

1. The nodes of the DAG consist of the strongly connected components of the call graph.
2. An arc exists between nodes C_I and C_J, $I \neq J$, if an arc exists in the call graph between two modules m_i and m_j, with $m_i \in C_I$ and $m_j \in C_j$.

Although each node of the DAG consists of a set of modules (those participating in a strongly connected component of the call graph), we treat these nodes as atomic entities. Any condition that holds for one module of a DAG node, such as residency in the logical address space, holds as well for all modules comprising that DAG node.

• *Generation of the overlay structure*

Nodes that lie along disjoint paths cannot be simultaneously active and can therefore be overlaid. Note that overlay generation is trivial if the DAG is a tree: Since no path exists between children of a node, the children can all be overlaid. If the DAG is more general than a tree, then complications arise due to the situation depicted in **Figure 2**.

From the above discussion, nodes *Y1* and *Y2* can be overlaid, since they lie on disjoint paths of the DAG. However, consider the placement of node *W*: The residency of node *W* is implied by the residency of *S1* or *S2*, since the execution of either node can result in executing node *W*. One solution would be to prohibit any overlaying of node *W*. By placing node *W* at the root of the DAG, where it could be invoked by any module, the requirements of *S1* and *S2* are satisfied, but the overlay structure is not maximal. (The call graph can always be augmented with an entry node that cannot be called from any other node.)

The GENOVLY algorithm performs a post-order traversal of the DAG. The algorithm transforms the DAG into a tree by moving offending nodes (such as node *W* in Figure 2) to a position farthest from the root that satisfies residency constraints. Consider the sample input to GENOVLY shown in **Figure 3**.

As in Figure 2, multiple arcs incident on node *W* imply the residency of that node for multiple paths through the call graph. The placement of node *W* is governed by the dominator relationship: Node *V dominates* node *W* if every path from the root to *W* contains *V*. The root dominates every node in the DAG, and so nodes such as *W* could properly be moved to the root, but this would preclude a maximum overlay structure. The dominators of a node *W* can be linearly ordered by their occurrence on a shortest path from the root to *W*. The dominator of *W* closest to *W* (other than *W* itself) is the *immediate dominator* of *W* [2]. A maximum overlay structure results from *absorbing* a node such as *W* into its immediate dominator *V*: Node *W* is removed from the tree proper, thus breaking its relationship with *S1* and *S2*. The modules of *W* are resident whenever the modules of *V* are resident, but the children of *W* are still eligible for overlaying, as are the children of *V*. This effect is shown as the output of GENOVLY in Figure 3.

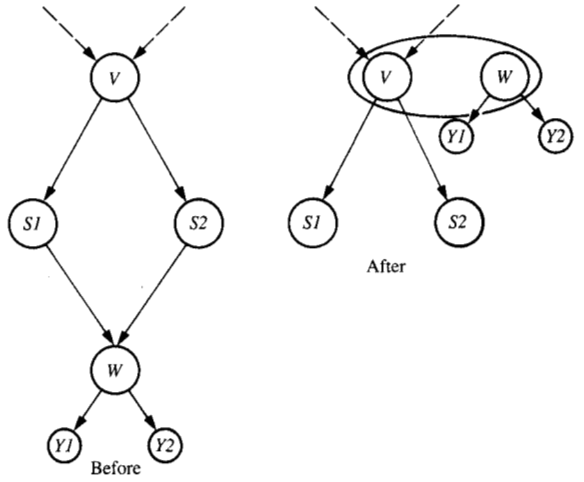

Figure 3

Input and output for GENOVLY algorithm.

The post-order traversal guarantees that when the GENOVLY algorithm places node *W*, it has been made the root of a subtree by absorbing any of its children with multiple predecessors into the relevant dominator. When node *W* is absorbed into node *V* in Figure 3, nodes *Y1* and *Y2* are undisturbed and remain children of node *W*.

Although the residency of node *W* must coincide with the residency of node *V*, it is still possible to overlay node *Y1* with *Y2*. However, the children of *W* and *V* cannot be treated alike: Nodes *S1* and *S2* are overlayable, and nodes *Y1* and *Y2* are overlayable, but the four nodes are not all overlayable.

At the conclusion of GENOVLY, the DAG has become a tree, since multiple arcs incident on a given node are deleted. Such nodes are absorbed into some other node that must still be in the tree.

```
Procedure GENOVLY (n)

absorbed (n) := ø;

For Each child ∈ Successors(n)
  GENOVLY (child);

/* Node n is now the root of a subtree */

If Indegree (n) > 1 Then
  d = Idom(n);
  absorbed(d) := absorbed(d) U {n}
                 U absorbed(n)
  /* Delete arcs incident on node n */
```

Figure 4

GENOVLY algorithm.

In **Figure 4**, we present a summary of the GENOVLY algorithm with the following notation:

Successors(*n*)	is the set of nodes S such that for each $s \in S$, $n \rightarrow s$ in the DAG.
Indegree(*n*)	is the number of arcs incident on node n in the DAG.
Idom(*n*)	is the immediate dominator of node n.
absorbed(*n*)	is the set of nodes absorbed into node n.

• *Emitting the overlay structure*

Given the tree constructed by GENOVLY and the *absorbed* sets, an overlay structure is easily produced. For completeness, this section shows how to emit an overlay structure in the style accepted by the IBM Linkage Editor [3]. The algorithm shown in **Figure 5** refers to the following functions:

CURSEG (v, h)	closes the currently open segment and opens an instance of segment (v, h). Code placed in different instances of segment (v, h) is overlaid. Code placed outside any segment is placed in the root. For the IBM Linkage Editor, CURSEG emits an *OVERLAY* statement with a name based on the values of v and h.
PLACE (n)	allocates the modules associated with node n to the current segment. For the IBM Linkage Editor, PLACE emits an *INSERT* statement for the modules of node n.

```
Procedure OUTPUT (n, v, h)

PLACE (n);

FOR EACH e ∈ absorbed(n)
  OUTPUT (e, v, h + 1);

FOR EACH child ∈ Successors(n)
  CURSEG ( v, h);
  OUTPUT (child, v + 1, h);
```

Figure 5

Generation of the overlay structure.

The algorithm is initialized by invoking OUTPUT (*root*, *0*, *0*). The algorithm first emits the modules associated with node n. These modules were previously identified as a strongly connected component of the call graph. Then, the OUTPUT procedure is called recursively for each node absorbed into node n. The modules associated with those nodes are all placed in the same segment instance as node n, but the children of the absorbed nodes are eligible for overlaying. Finally, a distinct instance of the same segment name is opened for each child of n, causing all children to be overlaid. Since the children themselves may have successors, OUTPUT is called recursively for each child.

• *Generating non-maximal overlay structures*

The algorithms presented above generate a maximal overlay structure for a program. If a specific bound on program size is desired, then the current size of the overlaid program could be maintained during the OUTPUT procedure. Once the program fits in the desired logical address space, all subsequent PLACE operations could place modules in the root segment. We do not claim this strategy to be optimal, because execution time depends on the frequency of overlay segment traffic as well as the size of the overlay segments.

3. Results

In **Table 1**, we show some results from applying the automatic overlay algorithm to three programs. Our tests were performed under the VM operating system [4] using the OS Linkage Editor [3]. The overlay algorithm generated the maximum overlay structure for the object code (data areas excluded) of each program. The test cases are as follows:

```
Procedure GENOVLY (n)

absorbed (n) := ø;

For Each child ∈ Successors(n)
  GENOVLY (child);

/* Node n is now the root of a subtree */

If Indegree (n) > 1 Then
  d = Idom(n);
  absorbed(d) := absorbed(d) U {n}
                 U absorbed(n)
  /* Delete arcs incident on node n */
```

Figure 4

GENOVLY algorithm.

In **Figure 4**, we present a summary of the GENOVLY algorithm with the following notation:

Successors(*n*)	is the set of nodes S such that for each $s \in S$, $n \rightarrow s$ in the DAG.
Indegree(*n*)	is the number of arcs incident on node n in the DAG.
Idom(*n*)	is the immediate dominator of node n.
absorbed(*n*)	is the set of nodes absorbed into node n.

• *Emitting the overlay structure*

Given the tree constructed by GENOVLY and the *absorbed* sets, an overlay structure is easily produced. For completeness, this section shows how to emit an overlay structure in the style accepted by the IBM Linkage Editor [3]. The algorithm shown in **Figure 5** refers to the following functions:

CURSEG (v, h)	closes the currently open segment and opens an instance of segment (v, h). Code placed in different instances of segment (v, h) is overlaid. Code placed outside any segment is placed in the root. For the IBM Linkage Editor, CURSEG emits an *OVERLAY* statement with a name based on the values of v and h.
PLACE (n)	allocates the modules associated with node n to the current segment. For the IBM Linkage Editor, PLACE emits an *INSERT* statement for the modules of node n.

```
Procedure OUTPUT (n, v, h)

PLACE (n);

FOR EACH e ∈ absorbed(n)
  OUTPUT (e, v, h + 1);

FOR EACH child ∈ Successors(n)
  CURSEG ( v, h);
  OUTPUT (child, v + 1, h);
```

Figure 5

Generation of the overlay structure.

The algorithm is initialized by invoking OUTPUT (*root*, *0*, *0*). The algorithm first emits the modules associated with node n. These modules were previously identified as a strongly connected component of the call graph. Then, the OUTPUT procedure is called recursively for each node absorbed into node n. The modules associated with those nodes are all placed in the same segment instance as node n, but the children of the absorbed nodes are eligible for overlaying. Finally, a distinct instance of the same segment name is opened for each child of n, causing all children to be overlaid. Since the children themselves may have successors, OUTPUT is called recursively for each child.

• *Generating non-maximal overlay structures*

The algorithms presented above generate a maximal overlay structure for a program. If a specific bound on program size is desired, then the current size of the overlaid program could be maintained during the OUTPUT procedure. Once the program fits in the desired logical address space, all subsequent PLACE operations could place modules in the root segment. We do not claim this strategy to be optimal, because execution time depends on the frequency of overlay segment traffic as well as the size of the overlay segments.

3. Results

In **Table 1**, we show some results from applying the automatic overlay algorithm to three programs. Our tests were performed under the VM operating system [4] using the OS Linkage Editor [3]. The overlay algorithm generated the maximum overlay structure for the object code (data areas excluded) of each program. The test cases are as follows:

Table 1 Results: Space is given in kilobytes; time is given in seconds.

Program	*Unoverlaid*			*Overlaid*		
	Space	*Time*	*Product*	*Space*	*Time*	*Product*
1. XX	9482	3.52	33376.6	8218	3.67	30160.0
2. PG	225	1.04	234.0	119	1.11	132.0
3. PR	7363	34.00	250342.0	3434	45.00	154530.0

XX is a language processor. Data areas comprise the bulk of the space, and these were not overlaid.

PG is a parser-generator.

PR is the Parafrase restructuring compiler from the University of Illinois at Urbana-Champaign [5]. The organization of this program is especially well suited to an overlay structure: One main routine could invoke fifty other routines along independent call chains.

The call graph of a program contains insufficient information to allow automatic overlaying of data areas. If a module depends on the persistence of data between invocations, then overlaying such data could change the semantics of the program. In the *XX* test case, data areas comprised the bulk of the logical address space, and the results of overlaying code did not significantly reduce the logical address space. However, in the *PG* test case, the author knew that the data areas could be overlaid. The space requirement then dropped to 110 kilobytes, and the program executed for 1.10 seconds with a space-time product of 121.0.

A sample of the overlay structure for the test case PR is shown in **Figure 6**.

```
/* Root Segment */
INSERT ANALYZE1
   .
   .
   .
   /* BLD_DD Absorbed
                  into Root */
   INSERT *BLD_DD1
      /* Called by BLD_DD: */
      OVERLAY V00H24
        INSERT *STRONG1
      OVERLAY V00H24
        INSERT *DDNAME1
      OVERLAY V00H24
        INSERT *DDARRA1
/* Long Sequence of
        Overlaid Modules */
OVERLAY V00H00
 INSERT ***INIT1
OVERLAY V00H00
 INSERT *#TIMER1
OVERLAY V00H00
 INSERT #REVPRT1
OVERLAY V00H00
 INSERT #SWPASS1
   .
   .
   .
```

Figure 6

Excerpt of the overlay structure for Parafrase.

Received August 26, 1985; accepted for publication June 20, 1986

Ron Cytron *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Cytron is a member of the PTRAN project in the Advanced Fortran group at the IBM Thomas J. Watson Research Center. An undergraduate of Rice

Acknowledgments

The authors wish to thank those at IBM Yorktown who provided help with the ideas and presentation of this paper, especially Philippe Charles, who provided the PG program, and Fran Allen, who provided many helpful comments.

References

1. R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.* **1**, No. 2, 146–160 (1972).
2. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1974.
3. *OS/VS Linkage Editor and Loader*, Order No. GC26-3813-6, 1983; available through IBM branch offices.
4. *VM/SP CMS Command and Macro Reference*, Order No. SC19-6209-2, 1983; available through IBM branch offices.
5. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Proceedings of CompSAC '80 (Fourth IEEE International Computer Software and Applications Conference)*, October 1980, pp.709–715.

University (B.S.E.E.), he attended graduate school at the University of Illinois at Urbana-Champaign, where he received an M.S. in 1982 and a Ph.D. in 1984 in computer science. Dr. Cytron's current interests include software for parallel processing, primarily automatic dependence analysis and restructuring techniques.

Paul G. Loewner *IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Loewner is an advisory programmer in the Computer Science Department, where he is working on the Research Queuing Package (RESQ) compiler and graphical workstation. He joined the Service Bureau Corporation (then a subsidiary of IBM) in 1958 and the Research Division in 1967. Mr. Loewner had previously worked on compiler construction with constant propagation, a linked data base subsystem, an automatic program documentation system (PLIDOC), graphics for speech system, radix exchange sort with lookahead, APL vs. PL/I interfacing problems, an interactive debugging system, and a generalized interprogram binding system. Mr. Loewner was a mathematician and programmer for Boeing Aircraft and North American Aviation from 1954 to 1957. He studied mathematics at Syracuse University from 1948 to 1951; at Stanford University from 1951 to 1953, receiving his B.S. with distinction in 1952 and his M.S. in 1953; at the University of California at Berkeley from 1953 to 1955; and at the Courant Institute of Mathematics, New York University, from 1958 to 1960. Mr. Loewner is a member of American Mathematical Society, the Association for Computing Machinery, the Mathematical Association of America, and the Society for Industrial and Applied Mathematics.