Semantics-Preserving Procedure Extraction [*]

Raghavan Komondoor and Susan Horwitz
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {raghavan, horwitz}@cs.wisc.edu

Abstract

Procedure extraction is an important program transformation that can be used to make programs easier to understand and maintain, to facilitate code reuse, and to convert "monolithic" code to modular or object-oriented code. Procedure extraction involves the following steps:

1. The statements to be extracted are identified (by the programmer or by a programming tool).
2. If the statements are not contiguous, they are moved together so that they form a sequence that can be extracted into a procedure, and so that the semantics of the original code is preserved.
3. The statements are extracted into a new procedure, and are replaced with an appropriate call.

This paper addresses step 2: in particular, the conditions under which it is possible to move a set of selected statements together so that they become "extractable", while preserving semantics. Since semantic equivalence is, in general, undecidable, we identify sufficient conditions based on control and data dependences, and define an algorithm that moves the selected statements together when the conditions hold. We also include a proof that our algorithm is semantics-preserving.

While there has been considerable previous work on procedure extraction, we believe that this is the first paper to provide an algorithm for semantics-preserving procedure extraction given an arbitrary set of selected statements in an arbitrary control-flow graph.

1 Introduction

Legacy code can often be improved by extracting out code fragments to form procedures (and replacing the extracted code with procedure calls). This operation is useful in several contexts:

- Legacy programs often have monolithic code sequences that intersperse the computations of many different tasks [LS86, RSW96]. Such code becomes easier to understand and to maintain if it is replaced by a sequence of calls (one for each task) [CY79]. This decomposition may also facilitate better code reuse [SJ87, LV97].
- When the same code appears in multiple places, replacing each copy with a procedure call makes the code easier to understand and to maintain (since updates need only be performed on a single copy of the code).
- A code fragment can sometimes be recognized as performing a (conceptual) operation on a (conceptual) object. Making that idea explicit by extracting the fragment into a procedure (or method) can make the code easier to understand, and can be an important part of the process of converting poorly designed, "monolithic" code to modular or object-oriented code [Par72].

For the purposes of this paper, extracting a procedure is defined by the following three steps:

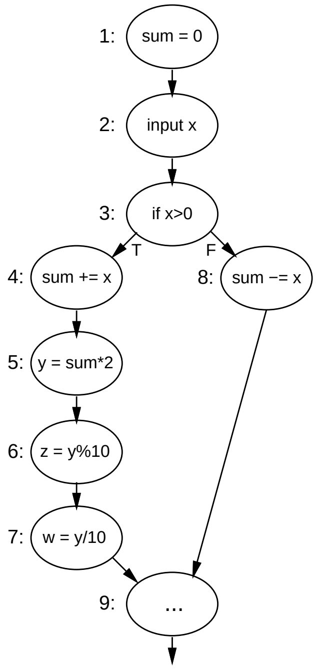
1. The statements to be extracted are identified.
2. If the statements are not contiguous, they are moved together so that they form a sequence that can be extracted into a procedure, and so that the semantics of the original code is preserved.
3. The statements are extracted into a new procedure, and are replaced with an appropriate call.

Although step 1 is very interesting, it is not the subject of this paper; we assume that the set of statements has been identified, either by the programmer or some kind of restructuring tool, such as those described in [LV97, BG98].

Step 3 involves deciding what the parameters to the procedure should be, which parameters should be passed by reference, and whether the procedure should return a value. These are straightforward issues (e.g., discussed in [LD98]).

Our interest is in step 2; in particular, we investigate the conditions under which it is possible to move a set of selected statements together so that they become "extractable", and so that semantics are preserved. To illustrate that this is a non-trivial problem, consider the following code fragment, represented by a control-flow graph (CFG):

[*] This work was supported in part by the National Science Foundation under grants CCR-9625656 and CCR-9970707, by the Microsoft Corporation, and by IBM.



Example 1: Although nodes 3 and 4 are contiguous in the CFG, they cannot be extracted into a procedure because of structural concerns: node 3 has an outgoing edge to node 8, and node 4 has an outgoing edge to node 5; if the two nodes are replaced by a call node (with a single successor in the CFG), either node 8 or node 5 will become unreachable, leading to a malformed CFG.

Example 2: Nodes 2 and 4 cannot be extracted into a procedure because of control dependence concerns. Moving node 4 before the *if* node would cause the wrong value to be assigned to variable *sum* when *x* is not positive; moving node 2 after the *if* node would cause the wrong value of *x* to be used to evaluate the condition. In either case, semantics would not be preserved.

Example 3: Nodes 4 and 6 cannot be extracted into a procedure because of data dependence concerns. This situation is similar to the one for nodes 2 and 4, discussed above: Moving node 6 before node 5 causes the wrong value of *y* to be used in the assignment to *z*, and moving node 4 after node 5 causes the wrong value of *sum* to be used in the assignment to *y*.

Example 4: Nodes 5 and 7 *can* be extracted. Node 7 can be moved up before node 6, which makes 5 and 7 an extractable sequence.

The chief contribution of this paper is an algorithm that moves a selected set of CFG nodes together so that they become extractable while preserving semantics. We believe that this algorithm is the first to handle an arbitrary set of selected nodes in an arbitrary (possibly unstructured) CFG. Since semantic equivalence is, in general, undecidable, it is not possible to define an algorithm for this problem that succeeds iff semantics-preserving procedure extraction can be performed for the given set of nodes. Therefore, we identify conditions based on control and data dependence that are *sufficient* to guarantee semantic equivalence; our algorithm succeeds iff those conditions hold. We also include a proof of correctness for our algorithm (that it performs only semantics-preserving procedure extraction).

A limitation of the algorithm is that it only moves CFG nodes; no duplication is performed. In Example 1, for instance, the indicated extraction can be done if the predicate is duplicated and the code restructured as follows:

```
if x>0  then sum += x
if x>0  then y = sum*2; z = y%10; w = y/10
        else sum -= x
```

Nevertheless, we feel that the algorithm is useful as is; in particular, it can be used as part of an automatic restructuring tool, and in that context failure of the algorithm can provide feedback indicating that code duplication is needed.

The remainder of the paper is organized as follows: Section 2 presents basic assumptions and terminology. Section 3 presents our algorithm for performing step 2 of procedure extraction. Section 4 discusses the algorithm's theoretical complexity, and Section 5 presents experimental results that give some insight into how well the algorithm will perform in practice. Section 6 states two theorems: the first theorem shows that the property we require to preserve control dependence is reasonable, and the second shows that the algorithm is correct (only performs semantics-preserving procedure extraction). Proofs for these theorems are included in an appendix. Section 7 discusses related work. Finally, conclusions are presented in Section 8.

2 Assumptions and Terminology

We assume that each procedure in a program is represented by a control-flow graph (CFG) that includes unique Enter and Exit nodes. Other CFG nodes represent predicates or simple statements (assignment, input, output, unconditional branch, or procedure call). A return statement is modeled as an unconditional branch to the Exit node. The Enter node is a special pseudo-predicate; it has two outgoing edges: one labeled "true" to the first statement or predicate in the procedure, and one labeled "false" to the Exit node (these edges are included so that all nodes in the body of the procedure are control dependence descendents of the Enter node). Normal predicate nodes also have two outgoing edges, one labeled "true" and the other labeled "false"; other nodes have one, unlabeled outgoing edge (for the purposes of this paper, there is no need to represent call/return connections among procedures; thus, a call node has one outgoing edge whose target is the statement or predicate that follows the call). Every node is reachable from the Enter node, and the Exit node is reachable from every node. $\mathcal{N}(G)$ denotes the nodes of a CFG G and $\mathcal{E}(G)$ denotes the edges.

We assume that the program includes no uses of uninitialized variables, and no assignments to dead variables. We also assume that appropriate static analyses (e.g., pointer analysis and interprocedural may-use, may-mod analysis) have been done so that the may-use and may-define sets are known for each CFG node (including call nodes).

For the purposes of this paper, two procedures are semantically equivalent iff when they are called in the same state (i.e., with the same mapping of variables – including the special stream variables *input* and *output* – to values), they finish in states that agree on the values of all variables that are (interprocedurally) live at Exit (with *output* considered to be live at all points in the program). A procedure that does not terminate, or that causes an exception – e.g., a division by zero – is considered to finish in the state in which all variables are mapped to $\perp$. Two CFGs are semantically equivalent iff the procedures that they represent are semantically equivalent.

We provide definitions of some standard concepts used in this paper:

Definition(*domination*) : CFG node p dominates node q iff all paths from Enter to q go through p. By definition, no node dominates itself. □

Definition(*postdomination*) : Node p postdominates node q iff all paths from q to Exit go through p. By definition, every node postdominates itself. □

Definition(*control dependence*) : Node p is C-control dependent on node q, where C is either "true" or "false", iff q is a predicate node, p postdominates the C-successor of q but it does *not* postdominate q itself.□

Definition(*flow dependence*) : Node p is flow dependent on node q due to a variable v iff v is used by p and defined by q and there is a CFG path from q to p that includes no node that must define v.□

Definition(*anti dependence*) : Node p is anti dependent on node q due to variable v iff v is used by p and defined by q and there is a CFG path from p to q.□

Definition(*output dependence*) : Node p is output dependent on node q due to variable v iff both nodes define v and there is a CFG path from q to p.□

Definition(*def-order dependence*) : Node p is def-order dependent on node q due to variable v iff both nodes define v, there is a node u that is flow dependent on both p and q due to v, and there is a CFG path from q to p.□

The definitions for flow, anti and output dependences are based on the definitions in [KKP$^+$81], while the definition of def-order dependence is from [BH93]. The following additional terms are also used in the paper:

Definition(*control dependence set*) : The *control dependence set* of node q in CFG G is the set of predicate-node, truth-value pairs (p, C) such that q is C-control dependent on p in G. □

Definition(*hammock*) : A *hammock* is a subgraph of a CFG that has a single *entry node*, and from which control flows to a single *outside exit node*. More formally: A hammock in CFG G is the subgraph of G induced by a set of nodes $H \subseteq \mathcal{N}(G)$ such that:

1. There is a unique entry node e in H such that: $(m \in \mathcal{N}(G) - H) \wedge (n \in H) \wedge ((m, n) \in \mathcal{E}(G)) \Rightarrow (n = e)$.
2. There is a unique outside exit node t in $\mathcal{N}(G) - H$ such that: $(m \in H) \wedge (n \in \mathcal{N}(G) - H) \wedge ((m, n) \in \mathcal{E}(G)) \Rightarrow (n = t)$.

□

Definition(*hammock chain*) : A *hammock chain* $(H_1, H_2, \ldots, H_m)$ in CFG G is a sequence of hammocks with no incoming edges from outside the sequence to any node other than H_1's entry node. That is, $\forall i \in [2 \ldots m]$:

1. the entry node of H_i is the outside exit node of H_{i-1}, and
2. $(n \in H_i) \wedge (m \in \mathcal{N}(G) - H_i) \wedge ((m, n) \in \mathcal{E}(G)) \Rightarrow (m \in H_{i-1})$

□

It can be seen that any hammock chain is itself a hammock. The entry node of the chain is the entry node of the first hammock H_1 and the outside exit node of the chain is the outside exit node of the last hammock H_m.

Definition(*atomic hammock*) : An *atomic hammock* is a hammock that is itself not a chain of smaller hammocks. □

It can be shown that a hammock H with entry node e is atomic iff for each hammock H_i that is strictly contained in H and also has entry node e, there exists a node n in $(H - H_i)$ such that there is an edge from n to e. This result is stated and proved as Theorem 3 in the appendix.

3 Semantics-Preserving Procedure Extraction

In this section we define an algorithm for reordering a given set of CFG nodes so that they can be extracted into a procedure while preserving semantics.

We assume the following inputs to the algorithm:

1. P, the control-flow graph of a procedure.
2. the set $\mathcal{M}$ of nodes in P that have been chosen for extraction ($\mathcal{M}$ is a subset of $\mathcal{N}(P)$ − {Enter, Exit})

The goal of the algorithm is to produce a CFG $P_{\mathcal{M}}$ that includes exactly the same nodes as P, so that:

- the nodes in $\mathcal{M}$ are extractable from $P_{\mathcal{M}}$, and
- $P_{\mathcal{M}}$ is semantically equivalent to P.

A very high-level description of our algorithm is as follows:

Step 1: Check whether the nodes in $\mathcal{M}$ are part of a chain of atomic hammocks in P; if not, then fail (P cannot be reordered to make the $\mathcal{M}$ nodes extractable while preserving control dependences).

Step 2: Create a polygraph that represents the ordering constraints imposed on the hammocks in the chain by data dependence considerations. (A polygraph is a graph with both "normal" edges and "either-or" edges. This will be clarified in Section 3.2 below.)

Step 3: Create the set of acyclic graphs defined by the polygraph created in Step 2.

Step 4: If any of the graphs created in Step 3 has a simple property (to be defined in Section 3.2), produce the corresponding CFG $P_{\mathcal{M}}$; otherwise, fail.

Subsection 3.1 explains the notion of extractability, and then step 1 of the algorithm, which concerns extractability and preserving control dependence. Subsection 3.2 elaborates on Steps 2–4, which have to do with preserving data dependence.

3.1 Extractability and Control Dependence

As stated earlier, a requirement is that the nodes in $\mathcal{M}$ be extractable from $P_{\mathcal{M}}$. What this means is that step 3 of procedure extraction – extracting the $\mathcal{M}$-nodes from $P_{\mathcal{M}}$ and replacing them with a procedure call node – does not result in a malformed CFG. In the example in the Introduction, nodes 5 and 7 can be moved together to result in a new CFG from which they are extractable, but this is not true for nodes 3 and 4 although they are already together. In essence, since the extracted procedure will have a single entry point, and the new procedure call node will have a single CFG successor, the set of $\mathcal{M}$-nodes must have the same two properties in $P_{\mathcal{M}}$ for the replacement to make sense: there must be a single $\mathcal{M}$-node that has incoming edges from outside the set, and all edges leaving the set must go to the same CFG node. It is thus easy to see that the nodes in $\mathcal{M}$ are extractable from $P_{\mathcal{M}}$ iff they form a hammock in $P_{\mathcal{M}}$.

Example 2 in the Introduction illustrates that a part of a sufficient condition to guarantee the semantic equivalence of P and $P_{\mathcal{M}}$ is that each node in $P_{\mathcal{M}}$ have the same control dependence set as in P.

Theorem 1 in the appendix shows that both of these objectives for the new CFG $P_{\mathcal{M}}$ – extractability and control dependence preservation – can be achieved iff in the original CFG P the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks.[1]

Every hammock in $\mathcal{C}$ must be either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$. Figure 1(a) illustrates this structure; the $\mathcal{M}$-hammocks are shaded. If we look back at Example 4 in the Introduction, $(4, 5, 6, 7)$ is the chain that contains nodes 5 and 7 with each of the nodes being an atomic hammock.

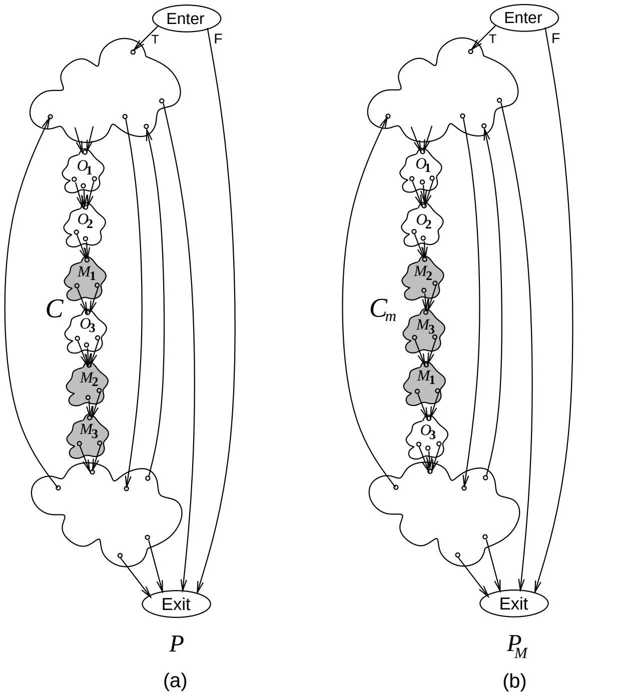


Figure 1: Chain containing $\mathcal{M}$ nodes

Algorithm Step 1 (finding chain $\mathcal{C}$)
Step 1 of our algorithm determines whether there is such a chain $\mathcal{C}$ in P as follows:

i. Identify the set of all hammocks in P:

```
for each node $p$ in $P$
    for each postdominator $q$ of $p$
        (a) do a depth-first search starting from
            $p$ and not going past $q$; let $H$ be the
            set of nodes visited by the depth-first
            search
        (b) $H$ is a hammock iff all edges coming
            into $H$ from outside $H$ have $p$ as their
            target.
```

ii. Eliminate from the set all hammocks that are neither $\mathcal{M}$-hammocks nor $\mathcal{O}$-hammocks (i.e., all hammocks that contain both $\mathcal{M}$-nodes and non-$\mathcal{M}$-nodes).

iii. Eliminate all non-atomic hammocks (see Section 2).

iv. Eliminate all non-maximal hammocks (a hammock is non-maximal if all of its nodes are contained in another hammock in the current set).

v. Check whether all $\mathcal{M}$-nodes are included in the final set of hammocks; if not, fail.

vi. Find the longest chain of atomic hammocks starting with any $\mathcal{M}$-hammock M_s as the current hammock: if the outside exit node of the current hammock is the entry node of some hammock H in the set, and all edges to this entry node from outside

[1] A part of the proof of the theorem also shows that P has such a chain iff the following two control-dependence conditions are met:

1. For every predicate node p in $\mathcal{M}$, all nodes that are control dependent on p are also in $\mathcal{M}$.
2. All nodes in $\mathcal{M}$ that are (directly) control-dependent on some node outside $\mathcal{M}$ have the *same* control dependence set outside $\mathcal{M}$.

We say that $\mathcal{M}$ is *well-formed in control dependence in P* in this case.

H come from the current hammock, then add H to the chain and make H the current hammock.

vii. Extend the chain backwards from M_s as far as possible: start with M_s as the current hammock; if the entry node of the current hammock is the outside exit node of some hammock H in the set, and all edges into the current hammock are from H, then add H to the beginning of the chain, and make H the current hammock.

viii. If all of the $\mathcal{M}$-hammocks are in the chain then chain $\mathcal{C}$ has been identified; if not, fail.

Example: For the CFG shown in Figure 1(a), step (ii) would eliminate from the set all hammocks that are neither $\mathcal{M}$-hammocks nor $\mathcal{O}$-hammocks (such as the entire CFG). It would leave in the set any O-hammocks that occur in the upper and lower "clouds" of code, and hammocks O_1, O_2, M_1, O_3, M_2, and M_3, as well as the non-atomic hammocks (O_1, O_2), and (M_2, M_3). Those non-atomic hammocks would be eliminated in step (iii). Any non-maximal hammocks inside the M_is or O_is or in the "clouds" would be eliminated in step (iv). Such hammocks arise, for example, in the context of a loop: the whole loop can be an atomic hammock, as well as the individual statements inside the loop; however, the individual statements are not maximal hammocks. Continuing with the example, step (v) would succeed, and step (vi) could start with M_1, M_2, or M_3. If it started with M_2, it would find the chain (M_2, M_3). Step (vii) would then extend that chain backward; the final chain would be: $(O_1, O_2, M_1, O_3, M_2, M_3)$.

3.2 Data Dependence

The goal of Steps 2–4 of our algorithm is to determine whether it is possible to permute the chain $\mathcal{C}$ into a chain $\mathcal{C}_m$ by reordering its hammocks so that:

- the $\mathcal{M}$-hammocks occur contiguously in $\mathcal{C}_m$, and
- the semantics of P are preserved.

If this can be done, then the CFG obtained by replacing $\mathcal{C}$ with $\mathcal{C}_m$ – shown in Figure 1(b) – is our desired CFG $P_{\mathcal{M}}$.

To preserve the semantics, we must ensure that on every execution, each node in P executes the same number of times and in the same states as in $P_{\mathcal{M}}$. It can be shown that permuting $\mathcal{C}$ in any way cannot alter the control dependence set of any node (see Part 3 of the proof of Theorem 1 in the appendix). Intuitively then, a permutation preserves semantics if it preserves the flow of values in the program: for each execution, if a node n in P uses a value defined at node m, then node n in $P_{\mathcal{M}}$ must also use the value defined at node m.

These observations lead us to define six kinds of ordering constraints (imposed by chain $\mathcal{C}$) that must be satisfied by the permutation. Some of the constraints are simple constraints of the form "hammock A must come before hammock B in the permutation." Others are "either-or" constraints of the form "either hammocks $A_1 \ldots A_j$ must all come before hammock B in the permutation, or hammock B must come before hammock C."

We can show that these 6 kinds of ordering constraints are sufficient: any permutation of the chain $\mathcal{C}$ that satisfies all of the constraints imposed by $\mathcal{C}$ preserves semantics. Theorem 2 in the appendix states this result formally, and a proof is included.

Algorithm Step 2 (building the polygraph) Step 2 of our algorithm involves building a *polygraph* that represents the ordering constraints imposed by the chain $\mathcal{C}$. Each node of the polygraph corresponds to one atomic hammock in $\mathcal{C}$; each edge in the polygraph represents one ordering constraint. A polygraph has two kinds of edges: *normal* edges (e.g., $A \rightarrow B$), which represent simple constraints, and *either-or* edges (e.g., $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$), which represent either-or constraints. A polygraph defines a *set* of (normal) graphs. Each graph in the set has the same nodes as the polygraph, and includes all of the polygraph's normal edges. For each either-or edge $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ in the polygraph, a graph in the set either includes the edges $A_1 \rightarrow B$, $A_2 \rightarrow B$, $\ldots$ $A_j \rightarrow B$, or it includes the edge $B \rightarrow C$. A polygraph that has k either-or edges thus defines a set of 2^k normal graphs.

The six kinds of ordering constraints are induced by instances of flow, def-order, anti, and output dependences between the hammocks in $\mathcal{C}$. The constraints are defined below and are illustrated using the chain shown in Figure 2, in which each node represents one atomic hammock, and the shaded nodes are $\mathcal{M}$-hammocks. Note that because of our assumptions that there are no uninitialized uses and no dead assignments, there must be a definition of variable v outside the chain that reaches chain entry, and v must be live at chain exit. For the purposes of this example, we will assume that the definition of v in hammock 8 does not reach the chain entry, and that x is not live at chain exit.

1. *Constraints induced by flow dependence*: Normal edge $A \rightarrow B$ is in the polygraph for chain $\mathcal{C}$ if

   (a) hammock A comes before hammock B in $\mathcal{C}$, and

   (b) there is a definition of a variable v in A that reaches a use of v in B.

   Example: For the chain in Figure 2, flow dependences induce the normal polygraph edges $1 \rightarrow 3$, $2 \rightarrow 3$, $3 \rightarrow 4$, $3 \rightarrow 5$, and $6 \rightarrow 7$.

2. *Constraints induced by def-order dependence*: Normal edge $A \rightarrow B$ is in the polygraph for chain $\mathcal{C}$ if

   (a) A comes before B, and

   (b) there are definitions of a variable v in both A and B, and

   (c) there is a use of v somewhere in the program that is reached by the definitions in both A and B.

   Example: Def-order dependences induce the normal polygraph edge $1 \rightarrow 2$ (because of the use of v in hammock 3).

3. *Constraints induced by anti dependence*: Normal edge $A \rightarrow B$ is in the polygraph for chain $\mathcal{C}$ if

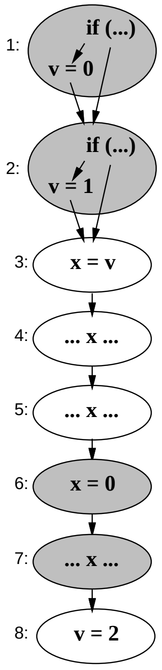


Figure 2: Chain used to illustrate ordering constraints (each node represents one atomic hammock)

(a) A comes before B, and

(b) there is a use of a variable v in A that is reached by a definition outside $\mathcal{C}$, and there is a definition of v in B.

Example: Anti dependences induce the normal polygraph edge 3 → 8 (because of the definition of v outside the chain that reaches chain entry).

4. *More constraints induced by anti dependence*: Either-or edge $\{A_1, A_2, \ldots, A_j\} \rightarrow B \parallel B \rightarrow C$ is in the polygraph for chain $\mathcal{C}$ if

   (a) C comes before any of the A hammocks, which all come before B, and

   (b) there is a non-empty set of variables V that are defined in both B and C, and

   (c) every A_i includes at least one use of a variable $v \in V$ that is reached by a definition in C, and

   (d) the set $A_1, ..., A_j$ is maximal: every hammock U that comes after C and before B, and that includes a use of a variable $v \in V$ that is reached by a definition in C is in the set.

   Example: Anti dependences induce the either-or polygraph edges $(\{4, 5\} \rightarrow 6 \parallel 6 \rightarrow 3)$, $(\{3\} \rightarrow 8 \parallel 8 \rightarrow 1)$, and $(\{3\} \rightarrow 8 \parallel 8 \rightarrow 2)$. However, note that the second and third edges are redundant, since normal edge 3 → 8 is also included in the polygraph because of a type 3 constraint.

5. *Constraints induced by output dependence*: Normal edge $A \rightarrow B$ is in the polygraph for chain $\mathcal{C}$ if

   (a) A comes before B, and

   (b) there is a definition of a variable v in both A and B, and

   (c) v is live at the exit of the chain $\mathcal{C}$, and

   (d) the definition in B reaches the exit of the chain but the definition in A does not reach the exit of the chain.

   Example: Output dependences induce the normal polygraph edges 1 → 8 and 2 → 8 (because v is live at chain exit).

6. *More constraints induced by output dependence*: Either-or edge $\{A_1, A_2, \ldots, A_j\} \rightarrow B \parallel B \rightarrow C$ is in the polygraph for chain $\mathcal{C}$ if

   (a) B comes before C, which comes before any of the A hammocks, and

   (b) there is a non-empty set of variables V that are defined in both B and C, and

   (c) every A_i includes at least one use of a variable $v \in V$ that is reached by a definition in C but is not reached by any definition in B, and

   (d) the set $A_1, ..., A_j$ is maximal: every hammock U that comes after C and that includes a use of a variable $v \in V$ that is reached by a definition in C but is not reached by any definition in B is in the set.

   Example: Output dependences induce the either-or polygraph edge $\{7\} \rightarrow 3 \parallel 3 \rightarrow 6$.

Note that because we have assumed that input and output are implemented using stream variables, there is no need to include special cases for constraints induced by I/O. For example, the statement *input x* is treated as both a use and a definition of the stream variable *input* (as well as a definition of x); therefore, if two hammocks in the chain each include an input statement, there will be a flow dependence from one hammock to the other, and the constraints defined above will ensure that the order of the input statements in the chain is maintained.

Example: The polygraph built for the example chain in Figure 2 is shown in Figure 3 (only the two non-redundant either-or edges are included). Normal edges are shown using plain arrows; either-or edge $\{A_1, A_2, ..., A_j\} \rightarrow B \parallel B \rightarrow C$ is indicated by enclosing the A_i nodes in a dashed circle, connecting that circle to node B with a heavy dashed arrow, connecting node B to node C with a second heavy dashed arrow, and linking the two dashed arrows with an arc.

Algorithm Step 3 (creating the acyclic graphs defined by the polygraph)
It is easy to see that a permutation $\mathcal{C}'$ of $\mathcal{C}$ satisfies all the constraints imposed by $\mathcal{C}$ iff $\mathcal{C}'$ is consistent with the edges of (at least) one of the graphs defined by the polygraph for $\mathcal{C}$ (which was created in step 2). Moreover, no permutation of $\mathcal{C}$ can be consistent with the edges of a cyclic graph defined by the polygraph. Therefore, to find permutations of $\mathcal{C}$ that satisfy all constraints imposed by $\mathcal{C}$, step 3 of our algorithm creates all acyclic graphs defined by the polygraph. This can be done with a recursive routine `CreateGraphs` whose inputs

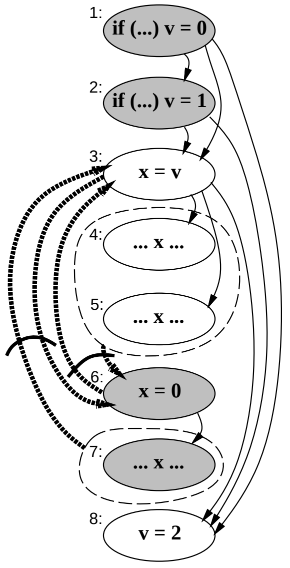


Figure 3: Polygraph built for the chain of Figure 2

are a graph G and a set of either-or edges E. The inputs to the top-level call of this routine are: a graph that contains all the polygraph's normal edges, and the set of all non-redundant either-or edges defined by the polygraph. The steps of the routine are:

1. If E is empty, return $\{G\}$ (G is one of the acyclic graphs defined by the polygraph).
2. Select an either-or edge $e = \{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ from E.
3. Let G' be equal to G augmented with edge $B \rightarrow C$; if G' is acyclic then call `CreateGraphs` recursively with G' and $E - \{e\}$ as inputs. Let $S1$ be the set returned by this recursive call.
4. Let G'' be equal to G augmented with edges $A_1 \rightarrow B, A_2 \rightarrow B, \ldots, A_j \rightarrow B$. If G'' is acyclic then call `CreateGraphs` recursively with G'' and $E - \{e\}$ as inputs. Let $S2$ be the set returned by this recursive call.
5. Return $S1 \cup S2$.

Example: Two acyclic graphs are defined by the polygraph shown in Figure 3. They are shown in Figure 4. Heavy dashed arrows indicate the edges that were added to account for the polygraph's either-or edges; plain arrows correspond to the polygraph's normal edges.

Algorithm Step 4 (creating the goal CFG $P_{\mathcal{M}}$) Step 4 of our algorithm determines whether it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable, without violating any of the constraints imposed by data dependence. This is accomplished by determining whether there is a permutation of the chain $\mathcal{C}$ in which all of the $\mathcal{M}$-hammocks are contiguous, and that is consistent with the edges of (at least) one of the acyclic graphs created in step 3. If such a permutation $\mathcal{C}_m$ is found, we can clearly produce $P_{\mathcal{M}}$ (in which all the $\mathcal{M}$ nodes form a hammock, and are thus extractable) by replacing $\mathcal{C}$ with $\mathcal{C}_m$ in P.

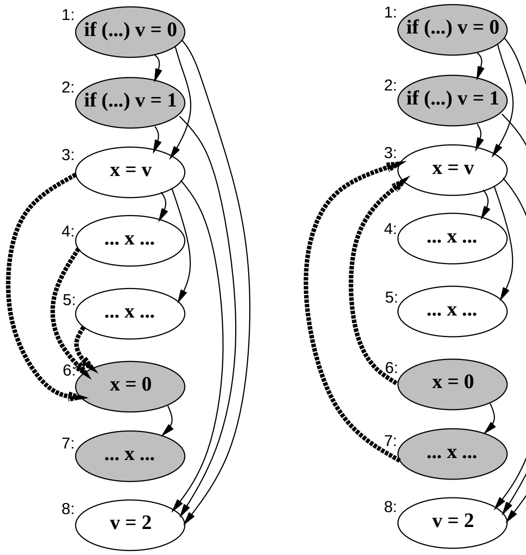


Figure 4: Acyclic graphs defined by the polygraph of Figure 3

It is easy to see that permutation $\mathcal{C}_m$ exists iff there is an acyclic graph G created in step 3 such that there are no paths in G that run from an $\mathcal{M}$-node to an $\mathcal{O}$-node to an $\mathcal{M}$-node[2] (the existence of such a path would preclude moving the endpoint $\mathcal{M}$-hammocks together).

This property can be easily checked for each graph G created in step 3 by using depth-first search as follows:

For each $\mathcal{O}$-node n in G:

1. Use depth-first search from n to determine whether there is a path to an $\mathcal{M}$-node.
2. If yes, use reverse depth-first search from n to determine whether there is also a path *from* an $\mathcal{M}$-node.
3. If yes, reject G

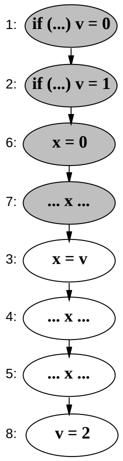
Once an acyclic graph G has been identified that has no such "illegal" paths, a final ordering $\mathcal{C}_m$ of the hammocks of the chain $\mathcal{C}$ can be produced by finding topological orderings of three subgraphs of G:

1. The subgraph induced by the set of $\mathcal{O}$-nodes from which there *is* a path to an $\mathcal{M}$-node.
2. The subgraph induced by the set of $\mathcal{M}$-nodes.
3. The subgraph induced by the set of $\mathcal{O}$-nodes from which there is *not* a path to an $\mathcal{M}$-node.

[2] Here we use $\mathcal{O}$-node to mean a node in G that represents an $\mathcal{O}$-hammock, and $\mathcal{M}$-node to mean a node in G that represents an $\mathcal{M}$-hammock.

The final ordering is the concatenation of the three topological orderings. Since G is acyclic, and there are no paths from an $\mathcal{M}$-node to an $\mathcal{O}$-node to an $\mathcal{M}$-node, the orderings are guaranteed to exist, and the concatenation is guaranteed to be consistent with the edges of G, thus satisfying all of the constraints imposed by the original chain $\mathcal{C}$.

Example: The first graph shown in Figure 4 would be rejected by step 4 of the algorithm because $\mathcal{O}$-nodes 3, 4 and 5 are all reachable from $\mathcal{M}$-nodes 1 and 2, and can reach $\mathcal{M}$-nodes 6 and 7. The second graph has no such illegal paths; in fact, there are no paths from an $\mathcal{O}$-node to a $\mathcal{M}$-node. Therefore, there are only two topological orderings to concatenate to form the final ordering (the ordering of the $\mathcal{M}$-nodes, followed by the ordering of the $\mathcal{O}$-nodes). One such final ordering is shown below.



4 Algorithm Complexity

The space complexity of the algorithm is polynomial in the length of chain $\mathcal{C}$, provided we make a minor change to the algorithm: instead of creating all the acyclic graphs in step 3 and then checking each of them in step 4, we need to merge the two steps. Thus, when an acyclic graph is created, step 4 should be applied on it. If the graph is rejected for having a bad path then it can be thrown away; otherwise, the algorithm can generate final ordering $\mathcal{C}_m$ and stop.

The time complexity of the algorithm is dominated by the combined steps 3 and 4. To see why, we begin by observing that finding the chain $\mathcal{C}$ (step 1 of the algorithm) takes time polynomial in the size of the CFG. Building the polygraph for chain $\mathcal{C}$ (step 2 of the algorithm) takes time polynomial in n, where n is the number of hammocks in $\mathcal{C}$ (n itself is polynomial in the size of the CFG).

The total time required to generate the acyclic graphs defined by the polygraph and find one without a bad path (combined step 3 and 4 of the algorithm) is determined by the total number of calls to the recursive routine `CreateGraphs` (described in Section 3.2). An individual call to this routine can make up to two recursive calls of its own, and therefore the total number of activations at any given depth k from the top-level activation is $\leq 2^k$. Since the maximum depth of recursion is equal to the number of either-or edges, which we refer to as e, and since $k \leq e$, the total number of calls to routine `CreateGraphs` is bounded above by $e \times 2^e$. Thus the time requirement of combined steps 3 and 4 is bounded above by $((e \times 2^e) \times (\text{some polynomial in } n))$. e itself is $O(n^2)$ in the worst case, as for any either-or edge $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$, there are $O(n^2)$ distinct pairs of hammocks for B and C and the set $A_1, A_2, \ldots A_j$ is fixed for any particular B and C.

We can improve this upper bound by recognizing that the number of activations of `CreateGraphs` at any given depth k from the top-level activation is also bounded above by $n!$ ($n!$ could be smaller than 2^k as k could be equal to e which itself is $O(n^2)$). The reason for this can be seen by considering the set of acyclic graphs received as inputs by the activations at this depth: each acyclic graph in the set has (at least) one permutation of $\mathcal{C}$ that is consistent with its edges, no permutation of $\mathcal{C}$ can be consistent with more than one graph in the set (because any two graphs differ on at least one either-or edge and no permutation can be consistent with both alternatives of an either-or edge), and there are only $n!$ different permutations of $\mathcal{C}$. Therefore an improved upper bound on the total time requirement of steps 3 and 4 is $((e \times \min(n!, 2^e)) \times (\text{some polynomial in } n))$.

This upper bound on the algorithm's time requirement is not surprising, since we have been able to show that the problem of finding a permutation $\mathcal{C}_m$ of $\mathcal{C}$ (if one exists) such that the $\mathcal{M}$ hammocks occur contiguously in $\mathcal{C}_m$, and $\mathcal{C}_m$ satisfies all the constraints imposed by $\mathcal{C}$, is NP–Hard in n. The NP–Hardness proof involves a reduction from the NP–Complete problem of determining whether a given schedule of database transactions is view-serializable (see [Pap86]).

Although this upper bound looks prohibitive, there is some evidence that the algorithm will work well in practice: We have measured n and e for all chains of atomic hammocks in a set of benchmark programs, and the results of the study (reported in the next section) are very encouraging. The bottom line is that in all of the programs, fewer than 1% of the chains have polygraphs with more than 5 either-or edges (i.e., have $e > 5$). Thus, it seems likely that the algorithm will usually have a reasonable running time.

Furthermore, the values of both n and e will be known by the end of step 2 (creating the polygraph), after doing work only polynomial in the size of the CFG; if both values are large, heuristics can be used in place of step 3. Two possible heuristics are:

1. Instead of generating the acyclic graphs defined by the polygraph, generate the permutations of $\mathcal{C}$ in which the $\mathcal{M}$-hammocks occur contiguously, and in which the relative ordering of the $\mathcal{M}$-hammocks and the relative ordering of the $\mathcal{O}$-hammocks are the same as in $\mathcal{C}$ (there are only $O(n)$ such permutations). For each generated permutation, check whether it satisfies the constraints of the polygraph. If so, use that permutation in place of $\mathcal{C}$ to obtain the CFG $P_{\mathcal{M}}$.

2. Limit the number of graphs defined by the polygraph by arbitrarily converting some or all of its

either-or edges to normal edges, then generate the corresponding acyclic graphs.

Of course, these heuristics will fail in some cases where the actual algorithm succeeds, but they may work well in practice.

5 Experimental Results

To provide some insight into the actual running time of our algorithm, we analyzed all chains of atomic hammocks in a set of benchmark programs. The steps carried out by the analysis for each program are listed below:

1. Use the SUIF compiler infrastructure front-end [WFW$^+$94] to build an intermediate form for the program.

2. Build a CFG for each procedure from its intermediate form.

3. Perform pointer analysis on the entire program. This provides information on the variables that might be pointed to by pointer variables, which in turn helps us in determining the variables that might be defined/used by statements that dereference pointers.

4. Compute summary information for each procedure. This information consists of the set of variables that might be defined, and the set of variables that might be used, as a result of a call to the procedure. This summary information is used to compute data dependences between call nodes and other nodes.

5. For each procedure in the program:

(a) Identify all atomic hammocks. Compute the may-use-before-defined set – the set of variables that might be used in a hammock before being defined – for each hammock by performing a backward dataflow analysis within the hammock. Compute the may-define set – the set of variables that may be defined by a hammock – for each hammock by unioning the may-define sets of all nodes in the hammock. Compute the must-define set of each hammock by unioning the must-define sets of all nodes in the hammock that postdominate the entry node.

(b) Identify all maximal-length chains of atomic hammocks in the CFG, as described in Algorithm Step 1 (in Section 3.1).

(c) For each chain, use the may-use-before-defined, may-define and must-define sets of the hammocks to compute the normal edges and either-or edges in the chain's polygraph.

The normal edges in the polygraph induce a transitive precedence relation on the hammocks in the chain. If A and B are hammocks in a chain, then the presence of a normal edge $A \rightarrow B$ implies that A precedes B in the relation. An either-or edge $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ is redundant if one of the A_i's precedes or is preceded by B, or if B precedes or is preceded by C. Every redundant either-or edge is eliminated and replaced by a set of normal edges; for instance if C is known to precede B, then $\{A_1, A_2, \ldots A_j\} \rightarrow B \parallel B \rightarrow C$ is replaced by the set of normal edges $\{A_1 \rightarrow B, A_2 \rightarrow B, \ldots, A_j \rightarrow B\}$.

Program	Number of procedures	Program size	
		No. of lines of source	Total No. of CFG nodes
agrep	65	6220	20725
allroots	6	449	724
anagram	16	655	1348
bc	101	8576	16146
bison-1.2.2	150	7852	27769
flex-2.4.7	147	8459	21975
football	57	2327	19383
gzip-1.2.4	99	7624	17886
ispell-4.0	121	7768	16484
simulator	110	5307	12049

Figure 5: Information about benchmark programs

Figure 5 gives some statistics for the benchmark programs we analyzed; *bc, bison, flex, gzip* and *ispell* are Gnu Unix utilities; *agrep* is described in [WM92]; *anagram* has been used in the experiments of [ABS94], while *allroots, football* and *simulator* were used in the experiments of [LRZ93].

Figure 6 gives the distribution of chains by length in each program. The taller bar gives the percentage of chains that have length ≤ 10, while the solidly shaded portion indicates the percentage of chains that have length ≤ 5. It can be observed that from 50 to 65% of chains have length ≤ 5. Figure 7 gives the cumulative number of chains over all programs for various chain lengths.

Figure 8 gives the distribution of chains by the number of either-or edges they have. The taller bar gives the percentage of chains that have ≤ 5 either-or edges, while the solidly shaded portion indicates the percentage of chains that have 0 either-or edges. It can be observed that from 91 to 99% of chains have no either-or edges, and virtually all the remaining ones have ≤ 5 edges. Figure 9 gives the cumulative number of chains over all programs that have a given number of either-or edges.

Recall that the algorithm's worst case time bound is proportional to $\min(n!, 2^e) \times$ (some polynomial in n) (where n is the length of the chain and e is the number of either-or edges in it). Our experimental results indicate that chain lengths tend to be short, and that the number of either-or edges tends to be very small; this is a strong indication that the running time of the algorithm will actually be polynomial in n for most chains in real programs, thus making it a feasible one in practice.

We also studied whether many either-or edges were made redundant by normal edges. The results were in the negative; in each program over 98% of the chains had $\leq$ 5 either-or edges even if redundant ones were *not* eliminated. We believe this further validates our

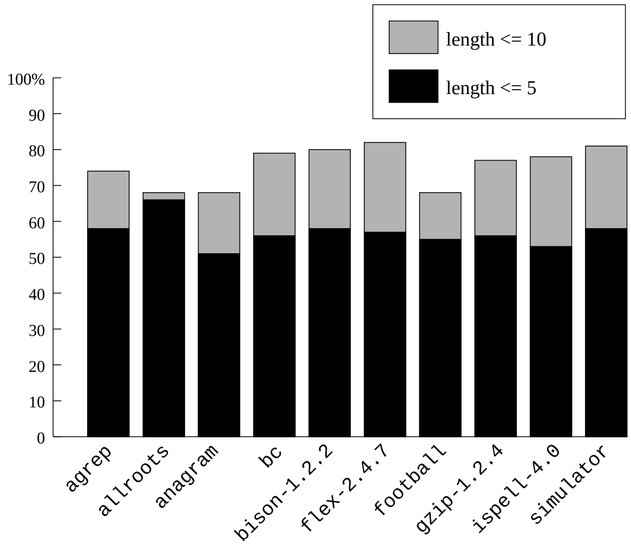


Figure 6: Distribution of chains by length

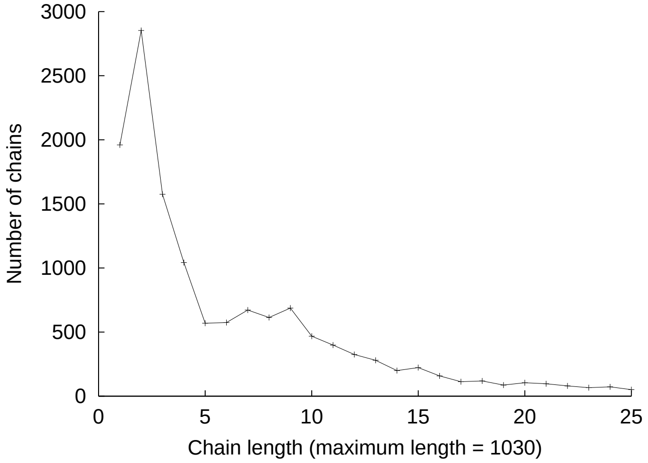


Figure 7: Chain lengths over all programs

conclusions.

Finally, Figure 10 gives for each program the total number of chains, length of the longest chain, and the maximum number of either-or edges in any single chain.

5.1 Factors Affecting the Experimental Results

Although our experimental results are quite encouraging, there are a number of factors that, if changed, might cause somewhat different results to be produced. These factors are discussed below.

Using the SUIF intermediate form: The SUIF intermediate form is a low-level representation. Since we build the CFG from the SUIF representation, each source level statement can correspond to many CFG nodes (this can be observed from the data in Figure 5). As an example, the single source level statement "*p++ = 0" becomes a series of CFG nodes that first save the original value of "p" into a temporary, then increment p, and finally store 0 into the location pointed to by the temporary. A result of this low-level representation is that we will tend to have longer chains than if we had worked at the source level, and we may have more normal and either-or edges per chain. On the other hand, the low-level representation has the advantage of flexibility; in the above example, it allows an extracted procedure to include just the increment of the pointer, or just the storing of the value 0.

Handling of pointers: For the sake of efficiency, we perform the *flow-insensitive* pointer analysis defined by Andersen [And94]. This in general gives less accurate results than a flow-sensitive analysis, which in turn could increase or decrease the number of normal and either-or edges in a chain.

Live Variable Analysis: Computing the normal edges

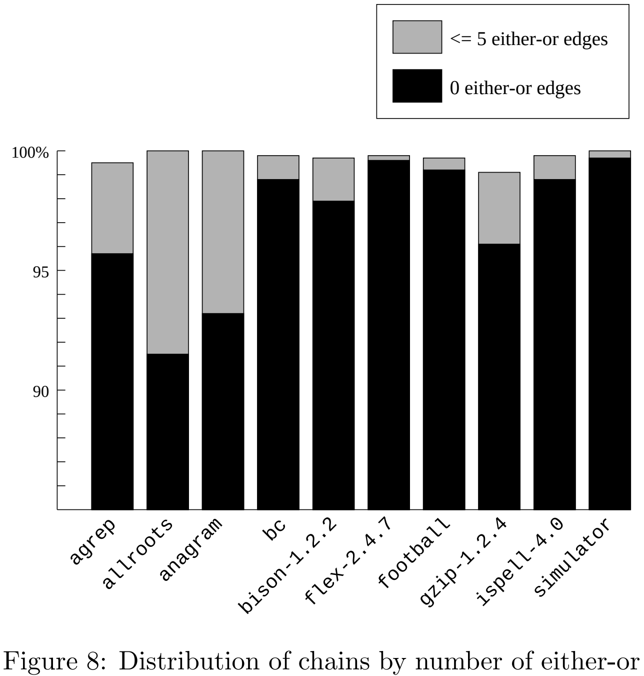


Figure 8: Distribution of chains by number of either-or edges (Note that the y-axis starts at 85%)

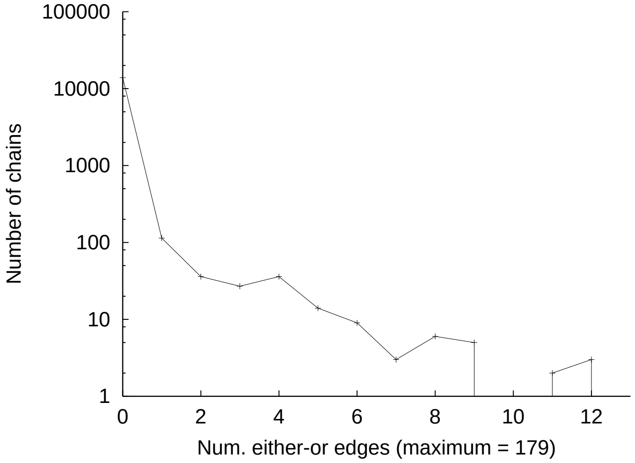


Figure 9: either-or edge counts over all programs

Program	Total number of chains	Longest chain	Max. num. of either-or edges
agrep	1821	99	86
allroots	47	93	3
anagram	104	70	4
bc	1514	169	8
bison-1.2.2	2487	80	35
flex-2.4.7	2151	194	179
football	1169	346	12
gzip-1.2.4	1680	127	133
ispell-4.0	1589	1030	6
simulator	1236	60	1

Figure 10: Measured statistics about benchmark programs

induced by output dependence (the fifth type of constraint described in Step 2 of the algorithm, Section 3) for a chain requires knowing which variables are live at chain exit. Rather than performing a whole-program live-variable analysis we assume that no variables are live at chain exit. This assumption can only increase the number of either-or edges reported for any chain.

Computing summary information for procedures: As mentioned earlier, we need to compute may-define and may-use sets for each procedure so that data dependence information for call nodes is known. We let the may-define (may-use) set of a procedure be equal to the set of variables that may be defined (used) by the procedure itself and by other procedures that could be (directly or indirectly) called by it.

We might have gotten different sets of normal and either-or edges if we had computed may-use-before-define summary information rather than just may-use information, but that would require an expensive whole-program analysis which we left out for the sake of efficiency.

Another issue is which variable are included in the may-define and may-use sets for a call node. Including all variables in the called procedure's may-define and may-use sets – even variables that are not visible to the calling procedure – can cause extra normal and either-or edges to be included in the polygraph (e.g., if a hammock chain includes two calls to the same procedure, and that procedure both defines and uses a local variable x, then there will be a flow dependence, and therefore a normal edge, between the two call nodes, since the definition of x at the first call will be considered to reach the use at the second call).

In our implementation we include in a call node's may-define (may-use) set all variables in the called procedure's may-define (may-use) set that are global, static, arrays, or structures, or whose addresses are taken. Including (local) structures is probably overly conservative; however, SUIF provides a single boolean function that identifies arrays, structures, globals, and variables whose addreses are taken; using that function provides a safe approximation to the set of variables in the called procedure's may-define and may-use sets that should be in the call node's sets.

Handling calls to library functions: Since source code for library functions is unavailable for analysis, we provided summary functions for most library functions. These summaries simulate definitions and uses of all variables that are not local to the library function (our summaries were based on the summaries that were used in the experiments reported in [WL95]). Use of different summary functions could lead to different sets of polygraph edges.

6 Key Theorems

Here we state two key theorems; proofs are given in the Appendix. The first theorem shows that it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable (they form a hammock) while preserving control-dependence sets for all nodes in P iff the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks in P. This demonstrates that Step 1 of our algorithm (which fails if the nodes in $\mathcal{M}$ are not part of a chain of atomic hammocks) is reasonable. The second theorem demonstrates that our algorithm is correct, by showing that any permutation of the hammocks in $\mathcal{C}$ that satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2) preserves semantics.

Theorem 1 *: Given CFG P and set of nodes $\mathcal{M}$, it is possible to create a new CFG P_e with the same nodes as P, with each node having the same set of control dependences as in P, and in which the $\mathcal{M}$ nodes form a hammock iff there is a chain $\mathcal{C}$ of atomic hammocks in P such that:*

1. *$\mathcal{C}$ includes all of the nodes in $\mathcal{M}$, and*
2. *every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.*

Theorem 2 *: Given:*

1. *Chain $\mathcal{C}$ of atomic hammocks in CFG P, such that all nodes in $\mathcal{M}$ are in the chain, and every hammock is either an $\mathcal{M}$-hammock (containing only*

$\mathcal{M}$ nodes) or an $\mathcal{O}$-hammock (containing no $\mathcal{M}$ nodes), and

2. *$\mathcal{C}'$, a permutation of $\mathcal{C}$ such that $\mathcal{C}'$ satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2), and*

3. *P', the CFG obtained by replacing chain $\mathcal{C}$ with the chain $\mathcal{C}'$ in P*

then: P and P' are semantically equivalent.

7 Related Work

Related work falls into two main categories: work related to procedure extraction (including [GN93, LD98, LV97, BG98]), and work on semantics-preserving transformations (including [BD77, LMW79, Ram88, PP96, Fea82, LV97, BG98, BDFH97, CLZ86, FOW87]).

[GN93] describes a tool that supports a set of meaning-preserving transformations on Scheme programs, including one that extracts a given contiguous sequence of expressions into a new function and replaces the sequence by a call to the function. If the user wishes to extract non-contiguous expressions into a function, the expressions must first be moved together using a transformation that moves a given expression to a given point if this does not change the program's meaning. The user is respnsible for identifying the correct program point and the correct ordering for the expressions. Our algorithm automates the aspect of bringing non-contiguous code together while preserving meaning. Additionally, our algorithm works on general unstructured programs whereas their transformations are limited to structured programs.

[LD98] also addresses the problem of procedure extraction, but their goal and approach are quite different from ours. Their goal is to discover and extract a meaningful computation surrounding a programmer-specified "seed" set of statements, within a programmer-specified bounding hammock. Their approach is to take the backward slice of the seed within the hammock, and then attempt to extract the slice as a procedure. The use of slices implicitly imposes the second part of our control dependence well-formedness condition. Their data dependence condition is quite restrictive: there can be no data flow from the extracted computation to the remaining computation in the bounding hammock, and vice versa, and no variable can be defined in both computations if the definitions reach uses outside the bounding hammock. In effect, they perform extraction only when there is no data dependence interaction between the code to be extracted and the rest of the code in the bounding hammock. In our approach, code can be extracted successfully in many situations where there are complex data dependence interactions between the extracted code and the remaining code. This flexibility comes at the price of a high worst-case time requirement; however, our experimental results indicate that this may not be a problem in practice. On the other hand, their approach allows predicate nodes to be duplicated, which may allow some procedure extractions that would not be possible using our approach. It may be possible to combine the advantages of both approaches: applying our procedure extraction algorithm on the backward slice of the seed will allow it to be extracted in some situations where their approach fails. Conversely, we might be able to make our data dependence condition more restrictive to reduce the time complexity of our algorithm, and to extend our algorithm to permit automatic code duplication when that is necessary to prevent the algorithm from failing.

Both [LV97] and [BG98] describe tools that identify sets of statements to be extracted into a procedure based on some user input. In [LV97], the programmer identifies a set of program variables as input variables, and another set of program variables as output variables. The tool then identifies the statements that make up the computation that defines the output variables using the values of the input variables. In the approach described in [BG98], the programmer specifies a set of variables, and the tool provides a visualization of the data flow graph of the computations that depend on or define any of the specified variables. The programmer then uses the visualization to select a "root" node, and the tool identifies the set of nodes on which the root node depends as a candidate for procedure extraction. Both of these tools can identify statements that are widely separated from each other or are not extractable unless the program is restructured to some extent. They do not address the extractability issue in general, however, and presumably leave actual extraction to be performed by the programmer. Our work is complementary to theirs in the sense that our procedure-extraction algorithm could be applied after one of these tools identifies the statements to extract.

Automatic transformations on programs are discussed in [BD77, Fea82, PP96, BDFH97]. [Fea82] proposes a system that accepts a set of recursion equations, a starting expression and a goal pattern. The system determines if there is a way to rewrite the starting expression into an expression that satisfies the goal pattern by a series of simple transformation steps, where each transformation step is a use of the recursion equations to perform an operation like folding, unfolding, instantiation or abstraction on the expression in hand. Although procedure extraction was not an explicit goal of these transformation systems, it is likely that similar techniques could be used to move the statements in $\mathcal{M}$ together via a series of simple meaning-preserving transformation steps.

Much has been reported in the literature about converting unstructured programs to structured programs by eliminating goto's (e.g.,[LMW79, Ram88]). Our work is related to this work in the sense that it involves meaning-preserving program transformations, although the goals of the transformations are different. In a similar sense, our work is related to work done on optimizing transformations such as code motion out of loops and other program regions (e.g., [CLZ86]). A notable similarity in fact exists between the necessary condition checked by the "strict" approach (one that guarantees an improvement in execution time) described in [CLZ86] and the condition checked by step 1 of our algorithm; they too stipulate that a node be moved out only if its control dependence ancestors can also be moved out.

[FOW87] mention the use of chains of hammocks in an approach to enable easy generation of sequential code from a Program Dependence Graph. They suggest factoring the control dependence subgraph of the PDG into hierarchical chains of hammocks, and say that code can be generated for a chain in any order consistent with the data dependences between the hammocks. Their concern however being program optimization and vectorization, they do not talk about procedure extraction or how the main problem therein of moving together a given set of nodes can be mapped to the problem of finding a certain chain and permuting it under a certain set of constraints.

8 Conclusions and Future Work

We have defined an algorithm that moves a selected set of nodes in a CFG together so that they become extractable while preserving program semantics. The algorithm places no restrictions on the structure of the CFG or on the selected set of nodes. Although the algorithm has a worst-case exponential time complexity, experimental results indicate that it may work well in practice.

The algorithm succeeds in moving the selected nodes together if and only if certain data and control dependence properties hold. These properties guarantee that semantics will be preserved; however they do not necessarily hold in all instances where semantics preserving extraction is possible. Our future research plans include studying how often the algorithm succeeds in extracting meaningful methods from real programs.

A Appendix: Correctness Proofs

This appendix includes proofs for the two theorems stated in Section 6. Theorem 1 shows that it is possible to move the nodes in $\mathcal{M}$ together so that they are extractable (they form a hammock) while preserving control-dependence sets for all nodes in P iff the nodes in $\mathcal{M}$ are part of a chain $\mathcal{C}$ of atomic hammocks in P. This demonstrates that Step 1 of our algorithm (which fails if the nodes in $\mathcal{M}$ are not part of a chain of atomic hammocks) is reasonable. Theorem 2 demonstrates that our algorithm is correct, by showing that any permutation of the hammocks in $\mathcal{C}$ that satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2) preserves semantics.

Finally, in Theorem 3, we state formally and prove the characterization for atomic hammocks mentioned in Section 2.

A.1 Necessity of a chain containing $\mathcal{M}$

Theorem 1 *: Given CFG P and a set of nodes $\mathcal{M}$, it is possible to create a new CFG P_e with the same nodes as P, with each node having the same set of control dependences as in P, and in which the $\mathcal{M}$ nodes form a hammock iff there is a chain $\mathcal{C}$ of atomic hammocks in P such that:*

1. *$\mathcal{C}$ includes all of the nodes in $\mathcal{M}$, and*
2. *every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.*

Proof:

The proof is in three parts. We first show that if CFG P_e with the specified properties exists, then the following two conditions must hold for $\mathcal{M}$ in P:

1. For every predicate node p in $\mathcal{M}$, all nodes that are control dependent on p are also in $\mathcal{M}$.
2. All nodes in $\mathcal{M}$ that are (directly) control-dependent on some node outside $\mathcal{M}$ have the *same* control dependence set outside $\mathcal{M}$. In other words, if q and s belong to $\mathcal{M}$, and both are control dependent on nodes outside $\mathcal{M}$, and p is some predicate node outside $\mathcal{M}$, then q is C-control dependent on p if and only if s is C-control dependent on p, where C is either "true" or "false".

We say that $\mathcal{M}$ is *well-formed in control dependence in* P in this case.

Next we show that if $\mathcal{M}$ is well-formed in control dependence in P, then a chain $\mathcal{C}$ with the specified properties exists in P.

Finally, we show that if the chain $\mathcal{C}$ exists in P then the CFG P_e can be obtained from P by permuting $\mathcal{C}$. Clearly the three parts together prove the theorem.

Before actually proving the three parts, we state without proof some lemmas that hold for every CFG.

Lemma 1
Let q be any CFG node that is neither the Enter node nor the Exit node, and let $S = [(p_1 = \text{Enter}) \rightarrow p_2 \rightarrow \cdots \rightarrow p_m \rightarrow q]$ be a path in the CFG (m could be equal to 1 in which case the path is simply $[(p_1 = \text{Enter}) \rightarrow q]$). There exists an integer $k, 1 \leq k \leq m$ such that:

1. p_k is a predicate node in S, and
2. the edge from p_k to its successor in S is labeled C, where C is either "true" or "false", and
3. q postdominates only the C-edge of p_k; i.e., q is C-control dependent on p_k, and
4. for all $l, k < l \leq m$: q postdominates p_l

□

Lemma 2
Let S be a CFG path from node p to node q such that:

1. q is control dependent on p, and
2. q postdominates all other nodes in S

Then, every node in S between p and q is a control dependence descendent of p. □

Lemma 3
Let p be a predicate node and let $\{q_1, q_2, \ldots, q_m\}$ be a set of nodes such that each q_i is C-control dependent on p.

1. There exists a total ordering on the q_i's imposed by the postdominance relation.
2. If there is a path in the CFG from the C-successor of p to q_i that does not include q_j, for any i and j such that $i \neq j$, then q_j postdominates q_i.

□

Lemma 4
Let H be a hammock.

1. If p is a node in H and q is a node outside H, then all paths from p to q include the outside exit node of H.
2. No node q outside H can be control dependent on a predicate node p inside H.

□

Lemma 5
Let H be a hammock in a CFG, p be a predicate node outside H, and q be a node in H. q is C-control dependent on p iff :

1. the entry node e_H of H is C-control dependent on p, and
2. q postdominates e_H

□

A.1.1 Proof Part 1

Given that there is a CFG P_e with the following properties:

1. P_e and the given CFG P have exactly the same set of nodes, and
2. the nodes in $\mathcal{M}$ form a hammock $H_{\mathcal{M}}$ in P_e (i.e. the $\mathcal{M}$ nodes are *extractable* from P_e), and
3. each CFG node has the same control dependence set in P_e as in P

the goal in this part is to show that $\mathcal{M}$ is well-formed in control dependence in P. The strategy is to first show that $\mathcal{M}$ is well-formed in control dependence in P_e, and then carry the result over to P.

First, we define some terms used in the proof:
Definition *(control dependence parent/ancestor/ descendent):* Node p is a *control parent* of node q in CFG G iff q is control dependent on p in CFG G. p is a *control ancestor* of q iff it is either a control parent of q or it is a control ancestor of some control parent of q. q is a *control descendent* of p iff p is a control ancestor of q. □

In the following lemmas, $e_{\mathcal{M}}$ is used to denote the entry node of hammock $H_{\mathcal{M}}$.

Lemma 6
Let p and q be CFG nodes such that $p \notin H_{\mathcal{M}}$ and $q \in H_{\mathcal{M}}$. Every path in P_e from p to q includes $e_{\mathcal{M}}$.

Proof:
This follows directly from the definition of a hammock – all edges from nodes outside $H_{\mathcal{M}}$ to nodes inside $H_{\mathcal{M}}$ come into $e_{\mathcal{M}}$. □

Lemma 7
$((p$ is a predicate node$) \wedge (p \in \mathcal{M}) \wedge (p$ is a control parent of q in $P_e)) \Rightarrow (q \in \mathcal{M})$.

Proof:
For contradiction, assume that p is a predicate node in $\mathcal{M}$ that is a control parent of q, but that q is not in $\mathcal{M}$. As all the $\mathcal{M}$ nodes are contained in $H_{\mathcal{M}}$, p must be in $H_{\mathcal{M}}$. Clearly q is outside $H_{\mathcal{M}}$, but this violates Lemma 4 part 2. □

Lemma 8
Let q_1 and q_2 be nodes in $\mathcal{M}$ and let p_1 and p_2 be nodes not in $\mathcal{M}$ such that q_1 is C_1-control dependent on p_1 and q_2 is C_2-control dependent on p_2 in P_e. It must then be true that q_1 is also C_2-control dependent on p_2 and q_2 is also C_1-control dependent on p_1 in P_e.

Proof:
Clearly q_1, q_2 are in $H_{\mathcal{M}}$ while p_1, p_2 are outside $H_{\mathcal{M}}$. We make some observations, all of which result from applying Lemma 5 on the given facts:

1. $e_{\mathcal{M}}$ is C_1-control dependent on p_1
2. q_1 postdominates $e_{\mathcal{M}}$ (the earlier point and this one are true because q_1 is C_1-control dependent on p_1)
3. $e_{\mathcal{M}}$ is C_2-control dependent on p_2
4. q_2 postdominates $e_{\mathcal{M}}$ (point 3 and this one are true because q_2 is C_2-control dependent on p_2)

From Lemma 5 and points 2 and 3 above it follows that q_1 is C_2-control dependent on p_2. From Lemma 5 and points 1 and 4 above it follows that q_2 is C_1-control dependent on p_1. □

Definition$(extDeps(q, P))$: If q is an $\mathcal{M}$-node in the CFG P, then $extDeps(q, P)$ is the set of pairs (p, C) such that $p \notin \mathcal{M}$ and q is C-control dependent on p in CFG P. □

A corollary of Lemma 8 follows:

Corollary 1
$((q_1 \in \mathcal{M}) \wedge (q_2 \in \mathcal{M}) \wedge (extDeps(q_1, P_e) \neq \{\}) \wedge (extDeps(q_2, P_e) \neq \{\})) \Rightarrow$
$(extDeps(q_1, P_e) = extDeps(q_2, P_e))$.

This corollary follows directly from Lemma 8. □

From Lemma 7 and Corollary 1 it follows that $\mathcal{M}$ is well-formed in control dependence in P_e. However, control dependence well-formedness is a property based only on the control dependence sets of nodes, and we know that each node has the same control dependence set in P_e as in P. We thus infer that $\mathcal{M}$ is also well-formed in control dependence in P.

A.1.2 Proof Part 2

Let $\mathcal{M}$ be well-formed in control dependence in P. Our goal is to show that there then exists a chain of atomic hammocks $\mathcal{C}$ in P such that:

1. $\mathcal{C}$ includes all of the nodes in $\mathcal{M}$.
2. Every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.

All lemmas in this part of the proof implicitly apply to the CFG P, and they assume $\mathcal{M}$ is well-formed in control dependence in P.

Lemma 9 is really a specialized version of Lemma 1.

Lemma 9
Let e be a node in $\mathcal{M}$. If
$S = [(p_1 = \text{Enter}) \to p_2 \to \cdots \to p_m \to e]$ is a path in the CFG such that $p_m \notin \mathcal{M}$ (m could be equal to 1 in which case the path is simply $[(p_1 = \text{Enter}) \to e]$), there then exists an integer $k, 1 \leq k \leq m$ such that:

1. p_k is a predicate node in S, and
2. the edge from p_k to its successor in S is labeled C, and
3. e postdominates only the C-edge of p_k; i.e., e is C-control dependent on p_k, and
4. $p_k \notin \mathcal{M}$, and
5. for all $l, k < l \leq m$: e postdominates p_l

Proof:
We consider two cases for p_m. If e *does not* postdominate p_m, then p_m must be a predicate node and e must be C-control dependent on p_m where C is the label on the edge $p_m \to e$. In this case, $p_k = p_m$ and we are done.

The other case is e postdominates p_m. Applying Lemma 1 on the path S, we know that there exists a predicate node p_k in S such that e is C-control dependent on p_k where C is the label of the outgoing edge from p_k in the path, and such that e postdominates every node on S between e and p_k. Now applying Lemma 2 on the suffix of path S from p_k to e, we infer that p_k is a control ancestor of p_m. Since $p_m \notin \mathcal{M}$ and $\mathcal{M}$ is well-formed in control dependence, no control ancestor of p_m – including p_k – can be in $\mathcal{M}$. We thus have our result. □

Definition(*entry node of* $\mathcal{M}$) : A node e in $\mathcal{M}$ is said to be an entry node of $\mathcal{M}$ iff there exists an edge $m \to e$ such that $m \notin \mathcal{M}$. □

Some corollaries of Lemma 9 are:

Corollary 2
For any entry node e of $\mathcal{M}$, $\mathit{extDeps}(e, P) \neq \{\}$.

Since e is an entry node of $\mathcal{M}$, there must exist a path from the Enter node of P to e such that the last node on the path before e is a node that does not belong to $\mathcal{M}$. Lemma 9 thus applies and tells us that there must exist a predicate node $\notin \mathcal{M}$ on which e is control dependent. □

Corollary 3
If e_1 and e_2 are any two entry nodes of $\mathcal{M}$, then: $\mathit{extDeps}(e_1, P) = \mathit{extDeps}(e_2, P) \neq \{\}$.

This follows from Corollary 2 and the definition of well-formedness in control dependence. □

Corollary 4
If $\{e_1, e_2, \ldots, e_m\}$ is the set of all entry nodes of $\mathcal{M}$ then there is a total ordering on this set imposed by the postdominance relation.

This follows from Corollary 3 and Lemma 3. □

Definition(*Region of an entry node*) : Let e be an entry node of $\mathcal{M}$. *Region*(e) is defined to be the set of nodes in $\mathcal{M}$ that can be reached from e along CFG paths that include only nodes in $\mathcal{M}$. By definition, $e \in \mathit{Region}(e)$. □

Lemmas 10 through 12 are used to prove Lemma 14, which says that the nodes in *Region*(e) constitute an $\mathcal{M}$-hammock in P.

Definition(*first node outside* $\mathcal{M}$ *from an* $\mathcal{M}$ *node*) : Let q be a node in $\mathcal{M}$. t is said to be a first node outside $\mathcal{M}$ from q if there is a path in P from q to Exit such that t is the earliest node in the path that does not belong to $\mathcal{M}$[3]. □

The following lemma with its corollaries says that control flows out of a region of $\mathcal{M}$ nodes to a unique node in the CFG.

Lemma 10
Let q be any node in $\mathcal{M}$. There is a unique node t in P such that t is a first node outside $\mathcal{M}$ from q.

Proof:
For contradiction, assume that t_1 and t_2 are two distinct nodes such that both are first nodes outside $\mathcal{M}$ from q. The following observations can be made:

1. There exists a path
   $S_1 = [(r_1 = q) \to r_2 \to \cdots r_m \to t_1]$ such that all the r_i's are in $\mathcal{M}$ (m could be equal to 1 in which case the path is simply $[(r_1 = q) \to t_1]$).
2. There exists a path
   $S_2 = [(s_1 = q) \to s_2 \to \cdots s_n \to t_2]$ such that all the s_i's are in $\mathcal{M}$ (n could be equal to 1).
3. There is a path S from the Enter node to q.
4. If t_1 is the Exit node of the procedure, then it postdominates q (the Exit node postdominates all nodes).

   We consider the case where t_1 is some node other than the Exit node. t_1 cannot be control dependent on any predicate in $\{(r_1 = q), r_2, \ldots, r_m\}$ as that would be a violation of the control dependence well-formedness condition. Therefore applying Lemma 1 on the path $S + S_1$ (the concatenation of paths S and S_1) we infer that t_1 postdominates all the nodes $\{(r_1 = q), r_2, \ldots, r_m\}$. In particular t_1 postdominates q.
5. By a similar argument t_2 postdominates q.

The fact that t_2 postdominates q combined with the fact that there is a path from q to t_1 that does not include t_2 implies that t_2 postdominates t_1. The fact that t_1 postdominates q combined with the existence of the t_1 free path from q to t_2 implies that t_1 postdominates t_2. However it is not possible for two CFG nodes to postdominate each other if the Exit node is reachable from all nodes. We thus have our contradiction. □

Definition(*outside*(q)) : *outside*(q) denotes the unique first node outside $\mathcal{M}$ from q. □

[3] every node in $\mathcal{M}$ must have a first node outside $\mathcal{M}$ since by definition, the Exit node does not belong to $\mathcal{M}$.

Some corollaries of Lemma 10 are:

Corollary 5
Let e be any entry node of $\mathcal{M}$. For any two nodes q_1 and q_2 in *Region*(e), *outside*(q_1) = *outside*(q_2).

This follows directly from Lemma 10 and the fact that q_1 and q_2 are both reachable from e through paths completely within $\mathcal{M}$. If q_1 and q_2 have different first nodes outside $\mathcal{M}$ then e would not have a unique first node outside $\mathcal{M}$. □

Corollary 6
outside(e) postdominates all nodes in *Region*(e).

For any node $q \in$ *Region*(e), by Lemma 10, all paths from q to Exit go through *outside*(q). This means *outside*(q) postdominates q. But by Corollary 5, *outside*(q) = *outside*(e). We hence get our result. □

Lemma 11
Let e_1 and e_2 be two distinct entry nodes of $\mathcal{M}$ such that e_2 postdominates e_1. Let d_2 be a predecessor of e_2 such that $d_2 \notin \mathcal{M}$. Then:

1. e_1 dominates d_2, and
2. there exists a path from e_1 to d_2 that does not include e_2.

Proof:
We prove both properties by showing that e_1 would postdominate e_2 if they did not hold. This of course contradicts the assumption that e_2 postdominates e_1.

1. Say e_1 does not dominate d_2. Then there is a path S from Enter to e_2 such that $d_2 \rightarrow e_2$ is the last edge in S and e_1 is not in S. Applying Lemma 9 there exists a predicate p in S such that the C-successor of p is in S, e_2 is C-control dependent on p, and $p \notin \mathcal{M}$. Applying Corollary 3, e_1 too is C-control dependent on p. Since the part of the path S from p to e_2 does not include e_1, we apply part 2 of Lemma 3 and infer that e_1 postdominates e_2.
2. Say e_1 dominates d_2, but all paths from e_1 to d_2 include e_2. Then for d_2 to be reachable from Enter, there would have to be a path from e_2 to d_2 that does not include e_1. Hence there is a cycle S_2 that includes the edge $d_2 \rightarrow e_2$ but not the node e_1. We consider two cases on d_2 and show that both cases lead to the contradiction:

   (a) e_2 is C-control dependent on d_2. Applying Corollary 3, e_1 too is C-control dependent on d_2. Clearly there is a path from d_2 to e_2 that does not include e_1. Therefore applying Lemma 3, part 2, we obtain the result that e_1 postdominates e_2.

   (b) e_2 is not control dependent on d_2; i.e. e_2 postdominates d_2. There must be a predicate p in S_2 such that e_2 is C-control dependent on p and such that e_2 postdominates all nodes in S_2 between p and e_2 (d_2 is included in these nodes). The reason such a node p must exist is that otherwise there would be no way to leave the cycle S_2, thus rendering the Exit node unreachable from e_2. Applying Lemma 2 on the segment of S_2 from p to e_2, we infer that p is a control dependence ancestor of d_2. Since $d_2 \notin \mathcal{M}$ and since $\mathcal{M}$ is well-formed in control dependence we infer that $p \notin \mathcal{M}$. Again, because $\mathcal{M}$ is well-formed in control dependence, we can use Corollary 3 to infer that e_1 too is C-control dependent on p. Now applying Lemma 3, part 2 (the lemma is applicable since the part of cycle S_2 from p to e_2 does not include e_1) we infer that e_1 postdominates e_2.

□

Lemma 12
Let e_1 and e_2 be two distinct entry nodes of $\mathcal{M}$. *Region*(e_1) and *Region*(e_2) have an empty intersection.

Proof:
For contradiction, assume the intersection is non-empty and let $p \in ($*Region*$(e_1) \cap$ *Region*$(e_2))$. Applying Corollary 5, we get *outside*(p) = *outside*(e_1) = *outside*(e_2).

One of e_1 or e_2 postdominates the other by Corollary 4, so without loss of generality let us assume e_2 postdominates e_1. Let d_2 be a CFG predecessor of e_2 such that d_2 is not in $\mathcal{M}$. By Lemma 11, e_1 dominates d_2 and there is a path S_1 from e_1 to d_2 that does not include e_2. Since e_2 postdominates e_1, e_2 postdominates all nodes in S_1. Let t be the first node on S_1 that is not in $\mathcal{M}$ (t could be d_2). Clearly t = *outside*(e_1). We have already shown that *outside*(e_1) = *outside*(e_2), and therefore t = *outside*(e_2). Applying Corollary 6 we infer that t postdominates e_2. But e_2 postdominates all nodes in S_1, and we thus have a contradiction where e_2 and t both postdominate each other. □

Lemma 13
For every node n in $\mathcal{M}$, there is an entry node e of $\mathcal{M}$ such that n is in *Region*(e).

Proof:
Let n be any node in $\mathcal{M}$. Let us consider a path S from Enter to n. Some suffix S_1 of this path must contain only nodes from $\mathcal{M}$. Let p be the first node in S_1. p cannot be the Enter node and therefore p is an entry node of $\mathcal{M}$. It is clear that n is in *Region*(p). □

Lemma 14
If e is an entry node of $\mathcal{M}$, then *Region*(e) is an $\mathcal{M}$-hammock, with e being its entry node and *outside*(e) being its outside exit node.

Proof:
We first show that all edges from nodes outside *Region*(e) to *Region*(e) come into e. For contradiction, say there is a node q such that $q \in$ *Region*(e) and $q \neq e$ and there exists a node $p \notin$ *Region*(e) such that there is a CFG edge from p to q. If $p \notin \mathcal{M}$ then q is an entry node of $\mathcal{M}$; this implies that $q \in ($*Region*$(q) \cap$ *Region*$(e))$, but this is impossible according to Lemma 12. We then consider the case where $p \in \mathcal{M}$. By Lemma 13, $p \in$ *Region*(e_1) where e_1 is some entry node of $\mathcal{M}$. We know $p \notin$ *Region*(e), therefore it follows that $e_1 \neq e$. Due to the edge from p to q, q also belongs to *Region*(e_1). This means that $q \in ($*Region*$(e) \cap$ *Region*$(e_1))$, which is a contradiction according to Lemma 12. We have thus shown that all edges from nodes outside *Region*(e) to

$\mathit{Region}(e)$ come into e.

All nodes reachable from e along paths that go through only $\mathcal{M}$-nodes belong to $\mathit{Region}(e)$ (by definition), and by Corollary 5 $\mathit{outside}(e)$ is the first node outside $\mathcal{M}$ for all nodes in $\mathit{Region}(e)$. Therefore all edges from nodes in $\mathit{Region}(e)$ to outside go to $\mathit{outside}(e)$.

Therefore $\mathit{Region}(e)$ is an $\mathcal{M}$-hammock, with e being its entry node and $\mathit{outside}(e)$ being its outside exit node. □

Definition(*Σ and Ord(Σ)*) : Σ is the set of all entry nodes of $\mathcal{M}$. $\mathit{Ord}(\Sigma)$ is the total order induced on Σ by the postdominance relation (Corollary 4). □

Definition(*Nodes in between consecutive regions*) : Let e_i and e_j be two distinct nodes in Σ such that e_j immediately follows e_i in $\mathit{Ord}(\Sigma)$. $\mathit{InBetween}(e_i, e_j)$ is defined to be the set of all nodes that are outside $\mathcal{M}$, and that can be reached from $\mathit{outside}(e_i)$ along CFG paths that include only nodes that are outside $\mathcal{M}$. By definition, $\mathit{outside}(e_i) \in \mathit{InBetween}(e_i, e_j)$. □

Lemmas 15 through 19 are used to prove Lemma 20, which says that $\mathit{InBetween}(e_i, e_j)$ is an $\mathcal{O}$-hammock.

Lemma 15
e_j postdominates all nodes in $\mathit{InBetween}(e_i, e_j)$

Proof:
By Lemma 12, e_j is not in $\mathit{Region}(e_i)$, and by Lemma 14, there is a path from e_i to $\mathit{outside}(e_i)$ that is completely within $\mathit{Region}(e_i)$ (i.e., that does not include e_j). This together with the fact that e_j postdominates e_i implies that e_j postdominates $\mathit{outside}(e_i)$.

By definition, all nodes in $\mathit{InBetween}(e_i, e_j)$ can be reached from $\mathit{outside}(e_i)$ without including e_j. Therefore, since e_j postdominates $\mathit{outside}(e_i)$, e_j postdominates all nodes in $\mathit{InBetween}(e_i, e_j)$. □

Definition(*direct path*) : Let e_i and e_j be in Σ (e_i could be equal to e_j). A *direct path* from $\mathit{outside}(e_i)$ to e_j is a CFG path from $\mathit{outside}(e_i)$ to e_j that includes none of the nodes in $\Sigma - \{e_j\}$. □

We now show some important properties of the nodes in Σ.

Lemma 16
If e_i and e_j are two distinct nodes in Σ such that e_j immediately follows e_i in $\mathit{Ord}(\Sigma)$, then:

1. there exists a direct path from $\mathit{outside}(e_i)$ to e_j, and

2. there does *not* exist a direct path from $\mathit{outside}(e_i)$ to any node in $\Sigma - e_j$, and

3. there does *not* exist a direct path from $\mathit{outside}(e_m)$ to e_j, where e_m is the last node in the sequence $\mathit{Ord}(\Sigma)$, and

4. $\mathit{outside}(e_i)$ dominates e_j

Proof:
Let e_l be any node in Σ such that e_l appears after e_j in $\mathit{Ord}(\Sigma)$. Let e_b be any node in Σ such that it is either equal to e_i or it appears before e_i in $\mathit{Ord}(\Sigma)$.

We derive some intermediate results that will be used to prove the four parts of the lemma.

i. there exists a path from $\mathit{outside}(e_i)$ to e_j, as, by Lemma 15, e_j postdominates $\mathit{outside}(e_i)$.

ii. there can be no direct path from $\mathit{outside}(e_i)$ to e_l. If we had a direct path from $\mathit{outside}(e_i)$ to e_l, then because of the fact that e_j does *not* postdominate e_l (e_j comes before e_l in $\mathit{Ord}(\Sigma)$), we may infer that e_j does *not* postdominate $\mathit{outside}(e_i)$; this however contradicts Lemma 15.

iii. $\mathit{outside}(e_i)$ postdominates e_b. The reason is that $\mathit{outside}(e_i)$ postdominates e_i (by Corollary 6), and e_i postdominates all nodes in Σ that appear before it in $\mathit{Ord}(\Sigma)$.

iv. we must have a path from $\mathit{outside}(e_i)$ to e_j that does not go through e_b. If such a path does not exist then e_b postdominates $\mathit{outside}(e_i)$ (because e_j postdominates $\mathit{outside}(e_i)$), and this contradicts our previous result that $\mathit{outside}(e_i)$ postdominates e_b.

v. we show that there can be no path S from $\mathit{outside}(e_i)$ to e_b that does not go through e_j. Assume for the sake of argument that such a path S exists. We already know there is a path from $\mathit{outside}(e_i)$ to e_j that does not include e_b. We thus infer:

    (a) Since e_b is the entry node of hammock $\mathit{Region}(e_b)$, we can assume there are no nodes in $\mathit{Region}(e_b)$ before e_b in S

    (b) Informally speaking there must be a branch point in S where the path to e_j separates; otherwise all paths from $\mathit{outside}(e_i)$ would have to go through e_j to reach e_b, or vice versa. Formally, e_b must postdominate exactly one of the outgoing edges – say, the one labeled C – of some predicate p that appears on S before e_b. Thus e_b is C-control dependent on p.

    (c) p has to be in $\mathcal{N}(P) - \mathcal{M}$, which implies $(p, C) \in \mathit{extDeps}(e_b, P)$. We prove this by contradiction. Assume $p \in \mathcal{M}$. By point (a) above, p is in $\mathit{Region}(e_c)$, where $e_c \in \Sigma$ and $e_c \neq e_b$. But this cannot be true as by Lemma 4, part 2, e_b cannot be control dependent on any node in $\mathit{Region}(e_c)$.

    (d) Since the prefix of path S up to p does not have e_j in it, e_j has to postdominate both outgoing edges of p for it to postdominate $\mathit{outside}(e_i)$. In other words, e_j is *not* control dependent on p and $(p, C) \notin \mathit{extDeps}(e_j, P)$.

    $\mathcal{M}$ being well-formed in control dependence, $\mathit{extDeps}(e_b, P) \neq \mathit{extDeps}(e_j, P)$ is not possible. Therefore there can be no path from $\mathit{outside}(e_i)$ to e_b that does not go through e_j, which means there can be no direct path from $\mathit{outside}(e_i)$ to e_b.

We now prove the four parts of the lemma.

1. By point i above there is a path from *outside*(e_i) to e_j, and by points ii and v this path does not go through any node in Σ before reaching e_j.

2. This part of the lemma follows from points ii and v above.

3. We prove this part by contradiction: assume there is a direct path S_1 from *outside*(e_m) to e_j. Let S be a path from the Enter node to *outside*(e_m). *outside*(e_m) postdominates e_j, the reason being: e_m postdominates e_j (e_m is either equal to e_j or it occurs after e_j in $Ord(\Sigma)$), and *outside*(e_m) postdominates e_m (Corollary 6). e_j thus cannot postdominate *outside*(e_m), and therefore an application of Lemma 1 on path $S + S_1$ tells us that e_j is C-control dependent on some node p after *outside*(e_m) in S_1. S_1 being a direct path, it follows from the definition that $p \notin \mathcal{M}$. We now apply Corollary 3 to infer that e_i too is C-control dependent on p. The suffix of S_1 from p to e_j does not include e_i, and therefore an application of Lemma 3, part 2, tells us that e_i postdominates e_j. This gives us our contradiction, as e_j coming after e_i in $Ord(\Sigma)$ means that it is e_j that postdominates e_i.

4. Let $D = \{d_j \mid ((d_j \to e_j) \in \mathcal{E}(P)) \wedge (d_j \in (\mathcal{N}(P) - \mathcal{M}))\}$. No CFG predecessor of e_j can be in *Region*(e_k), where $e_k \in \Sigma$ and $e_k \neq e_j$; if this were true then e_j would be in *Region*(e_k), but this is ruled out by Lemma 12. Therefore, by Lemma 13, any predecessor of e_j that is not in *Region*(e_j) is in D. This, together with the fact that *Region*(e_j) is a hammock with e_j as its entry node (Lemma 14), implies that all paths from Enter to any node in *Region*(e_j) include one of the nodes in D. By Lemma 11, part 1, e_i dominates all nodes in D. Therefore we infer that e_i dominates e_j. Since all paths from e_i to e_j include *outside*(e_i) by Lemma 4, part 1, *outside*(e_i) dominates e_j.

□

Let e_i and e_j be two distinct nodes in Σ such that e_j immediately follows e_i in $Ord(\Sigma)$.

The following lemma shows that all edges going out of *InBetween*(e_i, e_j) go to e_j.

Lemma 17
$(q \in InBetween(e_i, e_j)) \wedge (r \notin InBetween(e_i, e_j)) \wedge ((q \to r) \in \mathcal{E}(P)) \Rightarrow (r = e_j)$

Proof:
For contradiction assume that there is an edge $q \to e_k$ such that q is in *InBetween*(e_i, e_j), and e_k is some node that is not in *InBetween*(e_i, e_j) and that is $\neq e_j$. From the definition of *InBetween*(e_i, e_j), it is clear that e_k is an $\mathcal{M}$ node. Moreover, since $q \notin \mathcal{M}$, $e_k \in \Sigma$. By definition, q is reachable from *outside*(e_i) by a path that does not include e_j. Therefore there exists a direct path from *outside*(e_i) to e_k. By Lemma 16, *outside*(e_i) can have a direct path to no node in Σ other than e_j, thereby giving us a contradiction. We have thus shown that any edge out of *InBetween*(e_i, e_j) goes to e_j. □

The next lemma shows that there cannot be an edge from a node outside *InBetween*$(e_i, e_j) \cup$ *Region*(e_j) to node e_j.

Lemma 18
$((p \to e_j) \in \mathcal{E}(P)) \Rightarrow$
$(p \in (Region(e_j) \cup InBetween(e_i, e_j)))$

Proof:
For contradiction, assume that p is neither in *InBetween*(e_i, e_j) nor in *Region*(e_j), and that there is an edge $p \to e_j$.

We start by showing that p cannot be in $\mathcal{M}$. To prove this by contradiction, assume that p is in $\mathcal{M}$. Since it is not in *Region*(e_j), by Lemma 13 it must be in *Region*(e_l) where e_l is some entry node of $\mathcal{M}$ such that $e_l \neq e_j$. The edge $p \to e_j$ implies that e_j is in both *Region*(e_l) and *Region*(e_j), which contradicts Lemma 12. Thus, p is not in $\mathcal{M}$.

By Lemma 16, *outside*(e_i) dominates e_j. Since there is an edge $p \to e_j$, *outside*(e_i) also dominates p. Hence there must be a path from *outside*(e_i) to p. Since p is not in *InBetween*(e_i, e_j), by Lemma 17, any path S from *outside*(e_i) to p must first go to e_j; by Lemma 4 part 1, the path must then go to *outside*(e_j). Let *outside*(e_k) be the last node in S such that e_k is an entry node of $\mathcal{M}$ (e_k could be e_j, but it cannot be e_i). There can be no entry node e_l in S between *outside*(e_k) and p; if there were, then since p is outside $\mathcal{M}$ the path would have to go from e_l to *outside*(e_l) before reaching p (by Lemma 4) which contradicts the assumption that *outside*(e_k) is the last outside exit node in S before p.

Therefore the suffix of path S from *outside*(e_k) to p followed by the edge $p \to e_j$ gives us a direct path from *outside*(e_k) to e_j, where $e_k \neq e_i$. This contradicts Lemma 16 and we thus have our result. □

The next lemma shows that there cannot be an edge from a node outside *InBetween*$(e_i, e_j) \cup$ *Region*(e_i) to a node in *InBetween*(e_i, e_j).

Lemma 19
$((p \to q) \in \mathcal{E}(P)) \wedge (q \in InBetween(e_i, e_j)) \Rightarrow (p \in (Region(e_i) \cup InBetween(e_i, e_j)))$

Proof:
For contradiction, assume p is neither in *InBetween*(e_i, e_j) nor in *Region*(e_i), q is in *InBetween*(e_i, e_j), and there is an edge $p \to q$.

We start by showing that p cannot be in $\mathcal{M}$. To prove this by contradiction, assume it is in $\mathcal{M}$. By Lemma 13, p is in *Region*(e_l) where e_l is some entry node of $\mathcal{M}$. However, since $p \notin$ *Region*(e_i), we infer that $e_l \neq e_i$. q is not in $\mathcal{M}$ and therefore the edge $p \to q$ implies that $q =$ *outside*(e_l). e_j postdominates q by Lemma 15, and therefore there must be a path S_1 from q to e_j. S_1 includes no nodes in $\mathcal{M}$, as if it did then we would have a direct path from *outside*(e_i) through q to an entry node other than e_j, which is not possible according to Lemma 16. q being equal to *outside*(e_l), the path S_1 then is a direct path from *outside*(e_l) to e_j. This is a contradiction of Lemma 16, as we know that $e_l \neq e_i$. Thus we conclude that p is not in $\mathcal{M}$.

We now consider two cases for p. If e_i *does not* dominate p then there is a path S from Enter to e_j that includes p and q but not e_i. We apply Lemma 9 to infer that there is a predicate n on this path that is not in $\mathcal{M}$, and on which e_j is C-control dependent. As $\mathcal{M}$ is well-formed in control dependence, we infer that e_i too is C-control dependent on n. Then applying Lemma 3, part 2, we infer that e_i postdominates e_j which is a contradiction. We therefore consider the other case: e_i dominates p. Since $p \notin \mathit{Region}(e_i)$ we apply Lemma 4, part 1, and infer that $\mathit{outside}(e_i)$ dominates p. Now the argument is just as in the proof of Lemma 18. Any path from $\mathit{outside}(e_i)$ to p will have to include $\mathit{outside}(e_j)$ because p is outside $\mathit{InBetween}(e_i, e_j)$ and $\mathit{Region}(e_j)$. There is also a path from p to e_j through q that does not include any node in $\mathcal{M}$. Together we infer that there is a direct path from $\mathit{outside}(e_k)$ to e_j for some $e_k \neq e_i$. This cannot be true according to Lemma 16. □

Lemma 20
$\mathit{InBetween}(e_i, e_j)$ is an $\mathcal{O}$-hammock, with $\mathit{outside}(e_i)$ being its entry node and e_j being its outside exit node.

Proof:

1. By definition, no node in $\mathit{InBetween}(e_i, e_j)$ is in $\mathcal{M}$.
2. Lemma 19 says that for any edge to come in from a node p outside $\mathit{InBetween}(e_i, e_j)$ to a node q in $\mathit{InBetween}(e_i, e_j)$, p has to be in $\mathit{Region}(e_i)$. But all edges from $\mathit{Region}(e_i)$ to outside $\mathit{Region}(e_i)$ (q is outside $\mathit{Region}(e_i)$) go into the node $\mathit{outside}(e_i)$ (Lemma 14). Therefore any edge coming into $\mathit{InBetween}(e_i, e_j)$ from outside comes into node $\mathit{outside}(e_i)$.
3. By Lemma 17, all edges from a node in $\mathit{InBetween}(e_i, e_j)$ to a node outside $\mathit{InBetween}(e_i, e_j)$ go to node e_j.

□

Lemma 21
$(\mathit{Region}(e_i), \mathit{InBetween}(e_i, e_j), \mathit{Region}(e_j))$ is a hammock chain in CFG P.

Proof:
With Lemmas 14 and 20, we have shown that $\mathit{Region}(e_i)$, $\mathit{InBetween}(e_i, e_j)$ and $\mathit{Region}(e_j)$ are hammocks, the outside exit node of $\mathit{Region}(e_i)$ is the entry node of $\mathit{InBetween}(e_i, e_j)$, and the outside exit node of $\mathit{InBetween}(e_i, e_j)$ is the entry node of $\mathit{Region}(e_j)$. All that remains to be shown is that no spurious edges come into $\mathit{outside}(e_i)$ or into e_j:

- By Lemma 19, for any edge $p \rightarrow \mathit{outside}(e_i)$, p must either be in $\mathit{Region}(e_i)$ or in $\mathit{InBetween}(e_i, e_j)$.
- By Lemma 18, for any edge $p \rightarrow e_j$, p must either be in $\mathit{Region}(e_j)$ or in $\mathit{InBetween}(e_i, e_j)$).

□

We now show the existence of the chain $\mathcal{C}$ in P:

- If there is only one entry node $e \in \Sigma$, then $\mathit{Region}(e)$ is an $\mathcal{M}$-hammock that contains all the nodes in $\mathcal{M}$ (by Lemmas 14 and 13). $\mathit{Region}(e)$ is a hammock, and any hammock is by definition either an atomic hammock or a chain of atomic hammocks. Therefore the chain of atomic hammocks obtainable from $\mathit{Region}(e)$ is the chain $\mathcal{C}$.
- Let $\mathit{Ord}(\Sigma) = (e_1, e_2, \ldots e_m)$. Let e_i, e_j be two consecutive hammocks in $\mathit{Ord}(\Sigma)$. By Lemma 21, $(\mathit{Region}(e_i), \mathit{InBetween}(e_i, e_j), \mathit{Region}(e_j))$ is a hammock chain. Therefore it is clear that $(\mathit{Region}(e_1), \mathit{InBetween}(e_1, e_2), \mathit{Region}(e_2), \ldots, \mathit{InBetween}(e_{m-1}, e_m), \mathit{Region}(e_m))$ is a hammock chain where each $\mathit{Region}(e_i)$ is an $\mathcal{M}$-hammock and each $\mathit{InBetween}(e_i, e_j)$ is an $\mathcal{O}$-hammock. By Lemma 13, every $\mathcal{M}$ node is included in $\mathit{Region}(e_k)$ for some $k \in [1, 2, \ldots m]$. As observed earlier, any hammock is either itself atomic or is a chain of atomic hammocks. Therefore the chain of atomic hammocks obtainable from $(\mathit{Region}(e_1), \mathit{InBetween}(e_1, e_2), \mathit{Region}(e_2), \ldots, \mathit{InBetween}(e_{m-1}, e_m), \mathit{Region}(e_m))$ is the chain $\mathcal{C}$.

A.1.3 Proof Part 3

Let there exist a chain of atomic hammocks $\mathcal{C}$ in P such that:

1. $\mathcal{C}$ includes all of the nodes in $\mathcal{M}$.
2. Every hammock in $\mathcal{C}$ is either an $\mathcal{M}$-hammock – a hammock in which all nodes are in $\mathcal{M}$ – or an $\mathcal{O}$-hammock – one that has no nodes in $\mathcal{M}$.

Let $\mathcal{C}_m$ be a permutation of $\mathcal{C}$ (a chain is permuted by reordering its hammocks) such that the $\mathcal{M}$-hammocks of $\mathcal{C}$ occur contiguously in $\mathcal{C}_m$. Let $P_e = P\,[\mathcal{C}_m/\mathcal{C}]$ be the CFG obtained by replacing $\mathcal{C}$ in P by $\mathcal{C}_m$. Our goal in this part is to show that P_e has the following properties:

1. P_e and P have exactly the same set of nodes, and
2. the nodes in $\mathcal{M}$ form a hammock in P_e (i.e. the $\mathcal{M}$ nodes are *extractable* from P_e), and
3. each CFG node has the same control dependence set in P_e as in P

Since P_e is obtained from P by permuting the chain $\mathcal{C}$ in P, clearly the node sets of the two CFGs are the same. The $\mathcal{M}$-hammocks in $\mathcal{C}_m$ occur contiguously and hence form their own chain of hammocks $H_{\mathcal{M}}$ which is a subchain of $\mathcal{C}_m$. By definition, any chain of hammocks is itself a hammock, and therefore $H_{\mathcal{M}}$ is an $\mathcal{M}$-hammock (it may not be atomic) in P_e. We are given that all the $\mathcal{M}$ nodes are in $\mathcal{C}$ in P, and therefore all the $\mathcal{M}$ nodes are in hammock $H_{\mathcal{M}}$ in P_e.

We now show that each node has the same control dependence set in P_e as in P. We begin by stating a few lemmas.

Let $e_{\mathcal{C}}$ be the entry node of chain $\mathcal{C}$ and let $e_{\mathcal{M}}$ be the entry node of chain $\mathcal{C}_m$. Let t be the outside exit node of both chains.

Lemma 22

1. A path S that includes no nodes in $\mathcal{C}$ exists in P iff the same path S exists in P_e while including no nodes in $\mathcal{C}_m$.

2. A path S whose last node alone is in $\mathcal{C}$ (the last node will then be $e_{\mathcal{C}}$) exists in P iff a path S' exists in P_e such that S' is identical to S except that its last node is $e_{\mathcal{M}}$.

Proof:
This is true because P and P_e differ only in the two chains. □

Lemma 23
Let q be a node in $\mathcal{C}$. Therefore it is in $\mathcal{C}_m$ also. There is a path in P from $e_{\mathcal{C}}$ to t that does not include q iff there is a path in P_e from $e_{\mathcal{M}}$ to t that does not include q.

Proof:
q will be present on all paths from the entry node to the outside exit node iff q postdominates the entry node of the atomic hammock to which it belongs. This, together with the fact that $\mathcal{C}$ and $\mathcal{C}_m$ have the same set of atomic hammocks gives us the result. □

Lemma 24
Let p be a predicate node outside $\mathcal{C}$, q be any node other than Exit, and C be a condition, i.e. "true" or "false". There is a path in P from the C-successor of p to Exit that does not include q iff there is a path in P_e from the C-successor of p to Exit that does not include q.

Proof:

Let S be a path from the C-successor of p to Exit in P that does not include q. If S does not include any node in $\mathcal{C}$ then by Lemma 22, part 1, the same path exists in P_e.

If S does not include q but does include some nodes in $\mathcal{C}$, then it is composed of three subpaths:

1. S_1, from p to $e_{\mathcal{C}}$ such that $e_{\mathcal{C}}$ is the only node in S_1 that is in $\mathcal{C}$.
2. S_2, from $e_{\mathcal{C}}$ to t, through $\mathcal{C}$ (S_2 has to extend all the way to t because the Exit node – as it has no successors – may be equal to t but cannot occur inside $\mathcal{C}$)
3. S_3, from t to Exit such that S_3 includes no nodes from $\mathcal{C}$. S_3 could be empty.

None of the above subpaths include q. Now consider P_e. By Lemma 22, part 2, there is a path S_1' in P_e that is identical to S_1 except that its last node is $e_{\mathcal{M}}$. Hence S_1' does not include q ($e_{\mathcal{M}} \neq q$ because $e_{\mathcal{M}}$ occurs somewhere in S_2). By Lemma 23 we know there is a path S_2' in P_e from $e_{\mathcal{M}}$ to t that does not include q. S_3 carries over into P_e directly. Thus $S_1' + S_2' + S_3$ is the path in P_e from C-successor of p to Exit that does not include q.

We have thus shown that there is a path in P from the C-successor of p to Exit that does not include q implies that there is a path in P_e from the C-successor of p to Exit that does not include q. The implication in the other direction can be shown symmetrically. □

Let p be a predicate node and q be any node. We consider possible cases for p, and in each case show that q is C-control dependent on p in P iff it is C-control dependent on p in P_e.

1. p is in $\mathcal{C}$. Therefore p is in some atomic hammock H in $\mathcal{C}$.

   Let q be C-control dependent on p in P. Therefore by Lemma 4, q is also in H. Since $\mathcal{C}_m$ is a permutation of $\mathcal{C}$, H appears unchanged in $\mathcal{C}_m$, which implies that q is C-control dependent on p in P_e. It can be shown in a similar manner that q is C-control dependent on p in P_e implies that it is C-control dependent on p in P.
2. p is outside $\mathcal{C}$. Applying Lemma 24 gets us our result.

□

A.2 Ordering constraints guarantee semantics preservation

Theorem 2 (meaning preserving permutations)
Given:

1. *Chain $\mathcal{C}$ of atomic hammocks in CFG P, such that all nodes in $\mathcal{M}$ are in the chain, and every hammock is either an $\mathcal{M}$-hammock (containing only $\mathcal{M}$ nodes) or an $\mathcal{O}$-hammock (containing no $\mathcal{M}$ nodes), and*
2. *$\mathcal{C}'$, a permutation of $\mathcal{C}$ such that $\mathcal{C}'$ satisfies the ordering constraints imposed by $\mathcal{C}$ (as defined in Section 3.2), and*
3. *P', the CFG obtained by replacing chain $\mathcal{C}$ with the chain $\mathcal{C}'$ in P*

then: P and P' are semantically equivalent.

Proof:

The proof is centered around two key lemmas. Lemma 27 says that the sets of live variables at chain exit and at chain entry are the same for P and P'. Lemma 29 says that if control enters $\mathcal{C}$ in P and $\mathcal{C}'$ in P' in the same state (as defined in Section 2), then:

- at the entry point of every hammock H in $\mathcal{C}$ and $\mathcal{C}'$, the states in the two programs will be identical with respect to all variables that are upwards-exposed in H (variables that might be used in H before being defined), and
- at the chain exits, the states in the two programs will be identical with respect to all live variables.

Since P and P' differ only in the chains, these two lemmas can be shown to ensure identical semantics.

Let A be a hammock in $\mathcal{C}$ and $\mathcal{C}'$. Let $A.uses$ be the set of variables that have upwards exposed uses in A; i.e. the variables that could be used in A before being defined.

Definition(*PrecDefs*) : For any variable $v \in A.uses$, let $PrecDefs(A, v)$ be the sequence of hammocks that precede A in $\mathcal{C}$ and have definitions of v that reach A. That is, $PrecDefs(A, v) = (D_1, D_2, \ldots, D_k)$, where

1. a hammock H in $\mathcal{C}$ belongs to $\{D_1, D_2, \ldots, D_k\}$ iff H precedes A in $\mathcal{C}$ and has a definition of v that reaches the use in A.

2. D_i precedes D_{i+1} in $\mathcal{C}$ for each $i \in 1 \dots k-1$, but not necessarily immediately.

(It is possible that $\mathit{PrecDefs}(A, v)$ is an empty sequence.) In a similar manner let $\mathit{PrecDefs}'(A, v)$ be the sequence of hammocks that precede A in $\mathcal{C}'$ and have definitions of v that reach A. □

Lemma 25
For any hammock A in $\mathcal{C}$ and $\mathcal{C}'$ and for any variable v in $A.\mathit{uses}$, the two sequences $\mathit{PrecDefs}(A, v)$ and $\mathit{PrecDefs}(A, v)'$ are equal.

Proof:
We make a series of observations which together give us our result.

1. For each hammock D_i in $\mathit{PrecDefs}(A, v)$, there is an edge $D_i \rightarrow A$ in the polygraph of $\mathcal{C}$ induced by a flow dependence. Therefore, as $\mathcal{C}'$ satisfies $\mathcal{C}$, D_i will come before A in $\mathcal{C}'$.

2. Since every hammock in $\mathit{PrecDefs}(A, v)$ has a definition that reaches A, there is an edge $D_i \rightarrow D_{i+1}$ induced by a def-order dependence in the polygraph of $\mathcal{C}$, where D_i and D_{i+1} are any two consecutive hammocks in $\mathit{PrecDefs}(A, v)$. Thus the hammocks in $\mathit{PrecDefs}(A, v)$ occur in the same order in $\mathcal{C}$ and in $\mathcal{C}'$.

3. From the first two observations it is clear that the only way there could be a hammock D_i in $\mathit{PrecDefs}(A, v)$ but not in $\mathit{PrecDefs}'(A, v)$, is if there exists a different hammock that is not in $\mathit{PrecDefs}(A, v)$ but is in $\mathit{PrecDefs}'(A, v)$, and that "blocks" D_i's definition of v from reaching A in $\mathcal{C}'$. We prove that this cannot be the case by showing that no hammock that is not in $\mathit{PrecDefs}(A, v)$ can be in $\mathit{PrecDefs}'(A, v)$. This would wrap up the proof of the lemma.

   Let N be a hammock that is in $\mathit{PrecDefs}'(A, v)$ but not in $\mathit{PrecDefs}(A, v)$. N either has to occupy a position in $\mathcal{C}$ before the first hammock in $\mathit{PrecDefs}(A, v)$ or it has to be after A. Otherwise it would itself belong to $\mathit{PrecDefs}(A, v)$. We thus consider two possible cases, both of which lead to contradictions:

   (a) N comes before the first hammock in $\mathit{PrecDefs}(A, v)$ in $\mathcal{C}$. Since the definition of v in N does not reach the use in A in chain $\mathcal{C}$, we can infer two facts. Firstly, there is no v-definition free path through the first hammock D_1 in $\mathit{PrecDefs}(A, v)$, as otherwise it would not be the first hammock in $\mathcal{C}$ to have a definition of v that reaches A. Secondly, the constraints induced by output dependence will apply and will force N to either appear before D_1 or after hammock A in $\mathcal{C}'$. In the first case, D_1 would block the definition in N from reaching A in $\mathcal{C}'$ and thus N cannot belong to
   $\mathit{PrecDefs}'(A, v)$. In the second case it clearly cannot belong to $\mathit{PrecDefs}'(A, v)$.

   (b) N comes after A in $\mathcal{C}$. The argument here is similar to the one above except that it uses the constraints induced by anti dependence.

□

Definition(*variable v is upwards exposed in a chain*) : We say a use of a variable v in a hammock U is *upwards exposed* in chain $\mathcal{C}(\mathcal{C}')$ iff there is a v-definition free path from the entry node of $\mathcal{C}(\mathcal{C}')$ to the use in U. □

Lemma 26
For any hammock A in $\mathcal{C}$ and $\mathcal{C}'$ and for any variable $v \in A.\mathit{uses}$:

($\mathit{PrecDefs}(A, v)$ *may* define v) ⇔ ($\mathit{PrecDefs}'(A, v)$ may define v) ⇔ (the use of v in A is upwards exposed in chain $\mathcal{C}$) ⇔ (the use of v in A is upwards exposed in chain $\mathcal{C}'$)

Proof:
(($\mathit{PrecDefs}(A, v)$ may define v) ⇔ ($\mathit{PrecDefs}'(A, v)$ may define v)) is true because
$\mathit{PrecDefs}(A, v) = \mathit{PrecDefs}'(A, v)$ (by Lemma 25).

If ($\mathit{PrecDefs}(A, v)$ may define v) is true, then there can be no hammock that must define v and that is before the first hammock of $\mathit{PrecDefs}(A, v)$ in chain $\mathcal{C}$. Therefore, there is no hammock before A in $\mathcal{C}$ that must define v, which in turn implies that the use of v in A is upwards exposed in chain $\mathcal{C}$. Going the other way, if it is true that the use of v in A is upwards exposed in chain $\mathcal{C}$, then it is obvious that no hammock preceding A in $\mathcal{C}$ must defines v.

We can analogously show that (($\mathit{PrecDefs}'(A, v)$ may define v) ⇔ (the use of v in A is upwards exposed in chain $\mathcal{C}'$)). □

The following lemma says that same set of variables are live at chain entry in the two CFGs P and P'. The same is true at the chain exits. This property allows us to characterize the input/output behavior of the two chains.

Lemma 27
Let $e(e')$ be the entry node of the first hammock in $\mathcal{C}(\mathcal{C}')$. Let t be the outside exit node of the last hammock in $\mathcal{C}$. Note that t is also the outside exit node of the last hammock in $\mathcal{C}'$. The lemma has two parts:

1. The set of variables that are live just before t in CFG P is the same as the set of variables that are live just before t in CFG P'

2. The set of variables that are live just before e in CFG P is the same as the set of variables that are live just before e' in CFG P'

Proof:

We prove the first property. A variable v is live just before t in a CFG if there is a use of v somewhere in the program that can be reached by a (possibly interprocedural) path S from t such that S includes no nodes that must define v. If this use of v is outside $\mathcal{C}$ and $\mathcal{C}'$, then v is live before t in both P and P' for the following reason: the parts of S that are outside P/P' remain the same no matter which chain is used, and a v-definition free path through a chain implies the existence of a similar path through any permutation of the chain.

On the other hand if the use of v is in the chain, then Lemma 26 applies: the use of v must be upwards-exposed in both $\mathcal{C}$ and $\mathcal{C}'$. Thus the use is reachable from t in both CFGs which gives us our result.

For the second property, we know variable v can be live just before e in $\mathcal{C}$ for two reasons:

1. v is live just before t in P and there is a v-definition free path through $\mathcal{C}$, *or*

2. there is a use of v in a hammock that is upwards exposed in chain $\mathcal{C}$.

By the first part of the lemma the same variables are live just before t in both P and P'; we have also seen that a v-definition free path through a chain implies the existence of a similar path through any permutation of the chain. By Lemma 26 the same variables have upwards exposed uses in both chains. We thus have our result. □

An *evaluation* of a node in the CFG is the result of executing the node in some program state. For an assignment the evaluation is the value assigned to the left hand side variable, for a predicate it is the value of the predicate, for an input statement it is the value extracted from the input stream, and so on.

The following lemma states (without proof) the somewhat obvious fact that the execution behavior of a hammock H depends only on the values of the variables in $H.uses$ at the time control enters H.

Lemma 28
Let H be a hammock belonging to $\mathcal{C}$ and $\mathcal{C}'$. Let control enter H in the two CFGs P and P' with the same values for the variables in $H.uses$. It can then be shown that:

1. Control follows the same path of execution through H in both CFGs

2. Let n be a node in H that occurs (maybe more than once) in the path of execution through H. An execution of n in the path through H in CFG P evaluates identically as the corresponding execution of n in the path through H in CFG P'.

□

We now prove an important lemma, which says that the two chains have identical input/output behavior.

Lemma 29
Let S be a program state (as defined in Section 2) restricted to the variables that are live at the entry of chain $\mathcal{C}$ (or $\mathcal{C}'$). Let control enter chain $\mathcal{C}$ in CFG P in state S, and let it also enter chain $\mathcal{C}'$ in CFG P' in the same state S. It can then be shown that:

1. at the entry point of every hammock H in $\mathcal{C}$ and $\mathcal{C}'$, the states in the two programs will be identical with respect to all variables that are in $H.uses$, and

2. at the chain exits, the states in the two programs will be identical with respect to all variables live at that point.

Proof:
We first prove part 1 of the lemma. The proof is by induction on the position of H in the chain $\mathcal{C}$.
Base case: Let H be the first hammock in $\mathcal{C}$. It is clear that for any variable $v \in H.uses$, $\mathit{PrecDefs}(H, v)$ is empty. By definition the empty sequence *may define* v. Thus by Lemma 25, $\mathit{PrecDefs}'(H, v)$ is empty and there is no definition of v in $\mathcal{C}'$ that precedes H. Therefore, in both CFGs control enters H with v having the same value as it did at entry, which is $S.v$.
Inductive case: Let H be a subsequent hammock in $\mathcal{C}$. The inductive hypothesis is: if N is a predecessor of H in $\mathcal{C}$, then control enters N in both CFGs with the same values for variables in $N.uses$ (N of course need not be a predecessor of H in $\mathcal{C}'$).

Let v be any variable in $H.uses$. We consider two cases:

1. The last time v was assigned a value before control enters H in CFG P was before control entered chain $\mathcal{C}$.

   We show that the same is then true in CFG P'. For contradiction assume D is a hammock that precedes H in $\mathcal{C}'$ and that a definition of v in D was executed prior to control entering H in CFG P'. We make the following series of observations:

   (a) Since D is before H in $\mathcal{C}'$ and has a definition of v that reaches the use in H, $D \in \mathit{PrecDefs}'(H, v)$

   (b) By Lemma 25, D also belongs to $\mathit{PrecDefs}(H, v)$

   (c) Therefore D precedes H in $\mathcal{C}$ and it must have executed before control entered H in P

   (d) Applying the induction hypothesis and Lemma 28 control must have followed the same path through D in P as it did in P'. But then the same definition of v in D that was executed in CFG P' would have been executed in CFG P also. The last assignment to v in CFG P before control enters H would then have been *inside* chain $\mathcal{C}$, which is a contradiction.

   Therefore no definition of v was executed in any hammock preceding H in either CFG, which means control enters H in both CFGs with the value of v being equal to $S.v$.

2. In CFG P, v was last assigned a value i prior to entering H during the execution of a hammock D that precedes H.

   We show that the same is then true in CFG P'. Clearly D belongs to $\mathit{PrecDefs}(H, v)$. By Lemma 25, $D \in \mathit{PrecDefs}'(H, v)$. Therefore D comes before H in chain $\mathcal{C}'$ and it must have been executed before control entered H in CFG P'. The induction hypothesis applies to D, and using Lemma 28 we infer that v must have been assigned the value i before control leaves D in CFG P'. What remains to be shown is that no other hammock redefines v after control leaves D and before it enters H in CFG P'. In that case it can be shown that the same should have happened in CFG P which of course contradicts our starting assumption.

Therefore control enters H in both CFGs with the same value i for v, and this value was assigned to v by the same corresponding hammock D.

We now prove part 2 of the lemma. Let t be the outside exit node of both chains. Let us construct a new dummy hammock F such that $F.uses$ includes all variables that are live just before t. We state without proof the following property: given that $\mathcal{C}'$ satisfies $\mathcal{C}$, the chain $\mathcal{C}' + F$ satisfies the chain $\mathcal{C} + F$. $\mathcal{C}' + F$ is obtained by appending hammock F to the end of the chain $\mathcal{C}'$, and $\mathcal{C} + F$ is obtained analogously. For the sake of analysis, let us then append hammock F to the end of chain $\mathcal{C}$ in P and to the end of $\mathcal{C}'$ in P'. Using the property just stated and using part 1 of this lemma we infer that control enters F in both CFGs with the same values for all variables with upwards-exposed uses in F; i.e., all variables that are live at chain exit. This inference must clearly hold even if F were not there at the end of the two chains, and thus we show that control leaves $\mathcal{C}$ in P and in $\mathcal{C}'$ in P' with the same values for all variables that are live at that point. □

We now show informally that Lemmas 27 and 29 guarantee that P and P' are semantically equivalent; i.e, starting execution of P and P' in the same state S, both procedures end with the same values of the variables that are live at the Exit node. It is clear that if for some starting state the path of execution does not flow through the chain $\mathcal{C}$ in P, then the same is true in P'. That is because the two procedures are identical except for the chain. Therefore the state when control reaches the Exit node will be identical in the two CFGs.

If the path from Enter to Exit in P does flow through $\mathcal{C}$, then we can decompose the path into a sequence of subpaths as follows:

1. a path from Enter to e, the entry node of $\mathcal{C}$
2. a path through $\mathcal{C}$ from e to t, the outside exit node of $\mathcal{C}$
3. zero or more occurrances of the following sequence:
   (a) a path from t to e
   (b) a path through $\mathcal{C}$ from e to t
4. a path from t to Exit

Consider the execution path in P' for the same starting state S. It is clear that P' will initially follow a subpath from Enter to e', where e' is the entry node of $\mathcal{C}'$. Moreover this subpath is identical to subpath 1 above with identical changes resulting to the state. Control then follows a path through $\mathcal{C}'$ and this could be different from subpath 2 above. But Lemma 27 says that the set of live variables at chain entry is the same for both CFGs, and the same is true at chain exit. Using this fact and the Lemma 29, it is clear that although subpath 2 through the chain could be different in the two CFGs, in both cases control leaves the chain with the same values for the variables that are live at chain exit. Therefore, by repeating the argument on the rest of the execution path to Exit, it follows that execution reaches Exit in P and P' with identical values for variables live at that point. □

A.3 Characterization for atomic hammocks

Theorem 3 (Atomic hammock characterization) *A hammock H with entry node e is atomic iff for each hammock H_i that is strictly contained in H and also has entry node e, there exists a node n in $(H - H_i)$ such that there is an edge from n to e.*

Proof:

We first prove a property that holds for hammocks in general:

Lemma 30
Two different hammocks cannot have the same entry node and the same outside exit node.

Proof:
For contradiction, assume G and H are two different hammocks with the same entry node e and the same outside exit node t. One of the two hammocks must have a node that is not in the other. Assume without loss of generality that node g belongs to $(G-H)$. There has to be a path S from e to g that does not go through t (that is because g belongs to G, and every node in a hammock is reachable from the entry node through a path that does not go through the outside exit node). Let u be the earliest node in S that does not belong to H (u has to exist as g does not belong to H). Clearly $u \neq t$ as t is not there in S. But this means H has two outside exit nodes u and t which contradicts the definition of a hammock. □

Let e be the entry node of a hammock H, and let t be its outside exit node. If H is *not* atomic, then by definition it can be decomposed into a chain $(H_i, H_j, ...H_m)$. No hammock H_k, $j \leq k \leq m$, can have e as its outside exit node (as that would render the entire chain with no path to Exit). Therefore, H_i is strictly contained in H but no node in $(H - H_i)$ has an edge to e. We have thus shown that the characterization *does not* hold if H is non-atomic.

We now consider the case where H *is* atomic. For contradiction, assume that that the characterization does not hold; i.e., assume there is a hammock H_i with entry node e that is strictly contained in H such that there is no edge from any node in $R = (H - H_i)$ to e. We can then show that R is also a hammock, using a series of observations:

1. The outside exit node e_i of H_i belongs to R. The reason for this is that if it didn't it would then have to be equal to t, which implies that two different hammocks H and H_i have the same entry and outside exit nodes – an impossibility according to Lemma 30.
2. All edges coming into nodes in R have to come in from nodes in H. That is because e does not belong to R. Therefore all edges coming into R from outside R come from H_i. Since e_i is the outside exit node of H_i, we can infer that e_i is the only node in R that has edges into it from nodes outside R.
3. No node in R can have an edge to a node outside H unless the target node is t, as otherwise t would not be the unique outside exit node of H.

4. No node in R can have an edge to a node in H_i. This is because the target of such an edge would have to be e, but our starting assumption is that there is no edge from any node in R to e.

5. Points 3 and 4 imply that all edges from nodes in R to outside R go to node t.

6. Points 2 and 5 imply that R is a hammock with e_i as its entry node and t as its outside exit node.

In fact, (H_i, R) is a chain, since all edges coming to e_i from outside R come from H_i (by point 2 above). Therefore H can be decomposed into the chain (H_i, R) which contradicts the starting assumption that H is atomic. □

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, January 1977.

[BDFH97] J. A. Bergstra, T. B. Dinesh, J. Field, and J. Heering. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems*, 19(5):639–684, September 1997.

[BG98] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[CLZ86] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *ACM Symposium on Principles of Programming Languages*, pages 70–85, 1986.

[CY79] L. L. Constantine and E. Yourdon. *Structured Design*. Prentice–Hall, Englewood Cliffs, New Jersey, 1979.

[Fea82] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[GN93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[KH99] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. Technical report, Computer Sciences, University of Wisconsin-Madison, 1999.

[KKP$^+$81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[LD98] A. Lakhotia and J-C. Deprez. Restructuring programs by tucking statements into functions. *Information and Software Technology*, 40(11–12):677–689, November 1998.

[LMW79] R. C. Linger, H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice*. Addison–Wesley, Cambridge, Mass., 1979.

[LRZ93] W. Landi, B. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[LS86] S. Letovski and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, pages 198–204, May 1986.

[LV97] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–258, April 1997.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PP96] A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[Ram88] L. Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4):893–920, October 1988.

[RSW96] S. Rugaber, K. Stirewalt, and L. M. Wills. Understanding interleaved code. *Automated Software Engineering*, 3(1–2):47–76, June 1996.

[SJ87] H. M. Sneed and G. Jandrasics. Software recycling. In *Proc. Conf. Software Maintenance*, pages 82–90, 1987.

[WFW$^+$94] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29(12), pages 31–37, December 1994.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[WM92] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, January 1992.