Making Graphs Reducible with Controlled Node Splitting

JOHAN JANSSEN and HENK CORPORAAL
Delft University of Technology

Several compiler optimizations, such as data flow analysis, the exploitation of instruction-level parallelism (ILP), loop transformations, and memory disambiguation, require programs with reducible control flow graphs. However, not all programs satisfy this property. A new method for transforming irreducible control flow graphs to reducible control flow graphs, called Controlled Node Splitting (CNS), is presented. CNS duplicates nodes of the control flow graph to obtain reducible control flow graphs. CNS results in a minimum number of splits and a minimum number of duplicates. Since the computation time to find the optimal split sequence is large, a heuristic has been developed. The results of this heuristic are close to the optimum. Straightforward application of node splitting resulted in an average code size increase of 235% per procedure of our benchmark programs. CNS with the heuristic limits this increase to only 3%. The impact on the total code size of the complete programs is 13.6% for a straightforward application of node splitting. However, when CNS is used, with the heuristic the average growth in code size of a complete program dramatically reduces to 0.2%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; E.1 [**Data**]: Data Structures—*graphs*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Compilation, control flow graphs, instruction-level parallelism, irreducibility, node splitting, reducibility

1. INTRODUCTION

Many compiler optimizations such as data flow analysis, loop transformations, the exploitation of instruction-level parallelism, and memory disambiguation are simpler, more efficient, or only applicable when the control flow graph of the program is reducible.

In current computer architectures, improvements can be obtained by the exploitation of instruction-level parallelism (ILP). ILP is made possible due to higher transistor densities, which allows the duplication of function units and data paths. Exploitation of ILP consists of mapping the ILP of the application onto the hardware resources of the target architecture as efficiently as possible. This mapping is used for Very Long Instruction Word and superscalar architectures. The latter

Authors' address: Department of Electrical Engineering, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands; email: {johan; heco}@@cs.et.tudelft.nl.

are used in most workstations. These architectures may execute multiple operations per cycle. Efficient usage requires that the compiler fill the instructions with operations as efficiently as possible. This process is called scheduling. In order to find sufficient ILP to justify the cost of multiple function units and data paths, a scheduler should have a larger scope than a single basic block at a time. A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and always leaves at the end. Several scheduling scopes can be found which go beyond the basic block level [Hoogerbrugge and Corporaal 1994]. The most general scope currently used is called a *region* [Bernstein and Rodeh 1991]. A region is a set of basic blocks that corresponds to the body of a *natural* loop. Since loops can be nested, regions can also be nested in each other. Like natural loops, regions have a single entry point (the loop header) and may have multiple exits [Bernstein and Rodeh 1991]. In Hoogerbrugge and Corporaal [1994] a speedup over 40% is reported when extending the scheduling scope to a region; the problem of region scheduling is that it requires loops in the control flow graph with a single entry point. These flow graphs are called reducible flow graphs. Fortunately, most control flow graphs are reducible; nevertheless, the problem of irreducible flow graphs cannot be ignored. To exploit the benefits of region scheduling, irreducible control flow graphs should be converted to reducible control flow graphs.

Loop transformations require a clear nesting of loops and thus reducible flow graphs. Exploiting parallelism also requires efficient memory disambiguation. To accomplish this the nesting of loops must be determined. Since in an irreducible flow graph the nesting of loops is not clear, memory disambiguation techniques cannot be applied directly to these loops. To exploit the benefits of memory disambiguation, irreducible control flow graphs should be converted to reducible control flow graphs as well. Another pleasant property of reducible control flow graphs is the fact that data flow analysis, which is an essential part of any compiler, can be done more efficiently [Ryder and Paull 1986] without the use of special routines to handle irreducible flow graphs.

Related Work. The problem of converting irreducible flow graphs to reducible flow graphs can be tackled at the front-end or at the back-end of the compiler. In Erosa and Hendren [1994] and Ammarguellat [1992] methods for normalizing the control flow graph of a program at the front-end are given. These methods rewrite an intermediate program in a normalized form. During normalization, irreducible flow graphs are converted to reducible ones. To make a graph reducible, code has to be duplicated, which results in a larger code size. Since the front-end is unaware of the precise number of machine instructions needed to translate a piece of code, it is difficult to minimize the growth of the code size.

Another approach is to convert irreducible flow graphs at the back-end. The advantage is that when selecting what (machine) code to duplicate one can consider the resulting code size. Solutions for solving the problem at the back-end are given in Cocke and Miller [1969], Hecht [1977], Aho et al. [1988], and Cocke [1971]; all use a technique called node splitting. The solution given by Cocke and Miller [1969] and Cocke [1971] is very time complex and does not try to minimize the resulting code size. The method described by Hecht [1977] and Aho et al. [1988] is even more inefficient in the sense of minimizing the code size, but it requires less analysis.

Data flow algorithms prefer reducible flow graphs: the data flow algorithms discussed in Allen [1972], Hecht [1977], and Tarjan [1981] are only applicable to reducible flow graphs. The Graham-Wegman algorithm [Graham and Wegman 1976] can handle irreducible flow graphs at the cost of a loss in efficiency. Recently, algorithms have been proposed for exhaustive and incremental data flow analysis [Sreedhar et al. 1996b]. Although these algorithms can handle irreducible graphs, their complexity increases when handling them. Solving bidirectional data flow problems using elimination algorithms also requires reducible graphs [Dhamdhere and Patil 1993]. In this article a new method for converting irreducible flow graphs at the back-end is given which is very efficient in terms of the resulting code size.

Overview. In Section 2 reducible and irreducible control flow graphs are defined, and a method for the detection of irreducible flow graphs is discussed. The principle of node splitting and the conversion method described by Hecht et al., which is a straightforward application of node splitting, are given in Section 3. Our approach, Controlled Node Splitting (CNS), is described in Section 4. All known conversion methods convert irreducible flow graphs without minimizing the number of copies. With controlled node splitting it is possible to minimize the number of copies. Unfortunately, this method requires much CPU time; therefore a heuristic is developed that reduces the CPU time but still performs close to the optimum. This heuristic and the algorithms for controlled node splitting are presented. The results of applying CNS to several benchmarks are given in Section 5. Finally the conclusions are given in Section 6.

2. IRREDUCIBLE CONTROL FLOW GRAPHS

The flow of control of a program can be described with a control flow graph. A control flow graph consists of nodes and edges. The nodes represent a sequence of operations or a basic block, and the edges represent the flow of control.

Definition 2.1. The control flow graph of a program is a triple $G = (N, E, s)$, where (N, E) is a finite directed graph, with N the collection of nodes and E the collection of edges. There is a path from the initial node $s \in N$ to every node of the graph.

Figure 1 shows an example of a control flow graph with nodes $N = \{s, a, b, c, d, e, f\}$, edges $E = \{(s, a), (a, b), (a, c), (b, c), (c, d), (d, e), (d, f), (c, a), (e, c)\}$, and initial node s.

As stated in the introduction, several compiler optimizations require as input a reducible flow graph. Many definitions for reducible flow graphs are proposed. The one we adopt is given in Aho et al. [1988] and is based on the partitioning of the edges of a control flow graph G into two disjoint sets:

(1) The set *back edges* BE consists of all edges whose heads dominate their tails.

(2) The set *forward edges* FE consists of all edges that are not back edges; thus $FE = E - BE$.

A node u of a flow graph dominates node v, if every path from the initial node s of the flow graph to v goes through u. The dominance relations of Figure 1 are: node

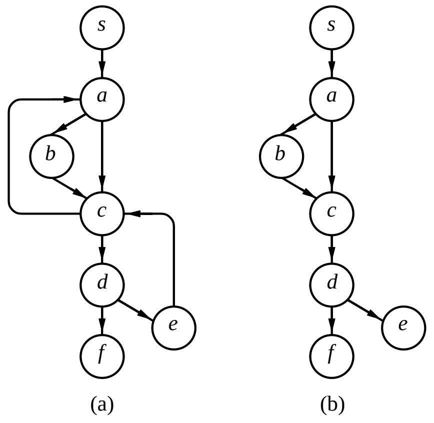


Fig. 1. (a) Reducible control flow graph; (b) the graph $G = (N, FE, s)$.

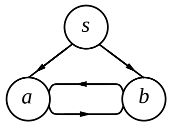


Fig. 2. The basic irreducible control flow graph.

s dominates all nodes; node a dominates all nodes except node s; node c dominates nodes c, d, e, f; and node d dominates nodes d, e, f. Therefore we have $BE = \{(c,a), (e,c)\}$ and $FE = \{(s,a), (a,b), (a,c), (b,c), (c,d), (d,e), (d,f)\}$. The definition of a reducible flow graph is as follows:

Definition 2.2. A flow graph $G = (N, E, s)$ is reducible if and only if its subgraph $G' = (N, FE, s)$ is acyclic and every node $n \in N$ can be reached from the initial node s.

The flow graph of Figure 1 is reducible, since $G' = (N, FE, s)$ is acyclic. The flow graph of Figure 2 is *irreducible*. The set of back edges is empty, because neither node a nor node b dominates the other. FE is equal to $\{(s,a), (s,b), (a,b), (b,a)\}$, and $G' = (N, FE, s)$ is not acyclic.

From Definition 2.2 we can derive that a control flow graph G is irreducible if the graph $G' = (N, FE, s)$ contains at least one loop. These loops are called irreducible loops. To remove irreducible loops, they must be detected first. There are several methods for doing this. Interval analysis as described in Allen [1972] and Allen and Cocke [1976] is one of them. Another method is described in Sreedhar et al. [1996a] which uses Tarjan's interval algorithm. All these methods are equally applicable. We have chosen the Hecht-Ullman T1-T2 analysis [Hecht and Ullman 1972; Ryder and Paull 1986]. This method is based on two transformations T1 and T2. These transformations are illustrated in Figure 3 and are defined as follows:

Definition 2.3. Let $G = (N, E, s)$ be a control flow graph, and let $u \in N$. The transformation $\mathrm{T1}(u)$ removes the self–loop edge $(u, u) \in E$, if this edge exists. The derived graph becomes $G' = \mathrm{T1}(u) = (N, E - \{(u,u)\}, s)$. In short, $G \overset{\mathrm{T1}(u)}{\Rightarrow} G'$.

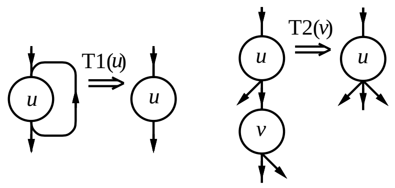


Fig. 3. The T1 and T2 transformation.

Definition 2.4. Let $G = (N, E, s)$ be a control flow graph, and let node $v \neq s$ have a single predecessor u. The transformation T2(v) is the consumption of node v by node u. The successor edges of node v become successor edges of node u. The original successor edges of node u are preserved except for the edge to node v. If I is the set of successor nodes of v, then the derived graph $G' = \mathrm{T2}\,(v) = (N - \{v\}\,, (E - \{(v,n) \mid n \in I\} - \{(u,v)\}) \cup\ \{(u,n) \mid n \in I\}\,, s)$. In short, $G \overset{\mathrm{T2}(v)}{\Rightarrow} G'$.

Definition 2.5. The graph that results when repeatedly applying the T1 and T2 transformations in any possible order to a flow graph, until a flow graph results for which no application of T1 or T2 is possible, is called the limit flow graph. This transformation is denoted as $\mathrm{T} = (\mathrm{T1} \mid \mathrm{T2})^*$.

In Hecht [1977] it is proven that the limit flow graph is unique and independent of the order in which the transformations are applied.

THEOREM 2.6. *A flow graph is reducible if and only if the limit flow graph contains a single node.*

The proof of this theorem can be found in Hecht and Ullman [1972]. An example of the application of the T1 and T2 transformations is given in Figure 4. The flow graph from Figure 1 is reduced to a single node, so we can conclude that this flow graph is reducible.

If after applying T1 and T2 transformations the resulting flow graph consists of multiple nodes, the graph is irreducible. The transformations T1 and T2 not only detect irreducibility, but they also detect the nodes that cause the irreducibility. Examples of irreducible graphs are given in Figure 5. From Theorem 2.6 it follows that a flow graph is irreducible if and only if the limit flow graph is not a single node.[1]

3. FLOW GRAPH TRANSFORMATION

If a control flow graph appears to be irreducible, a graph transformation technique can be used to obtain a reducible control flow graph. In the past some methods were given to solve this problem [Aho et al. 1988; Cocke and Miller 1969; Hecht 1977]. Most methods for converting an irreducible control flow graph are based on a technique called node splitting. This technique is described in Section 3.1. Section 3.2

[1]Another definition, which is more intuitive, is that a flow graph is irreducible if it has at least one loop with multiple loop entries [Hecht and Ullman 1972].

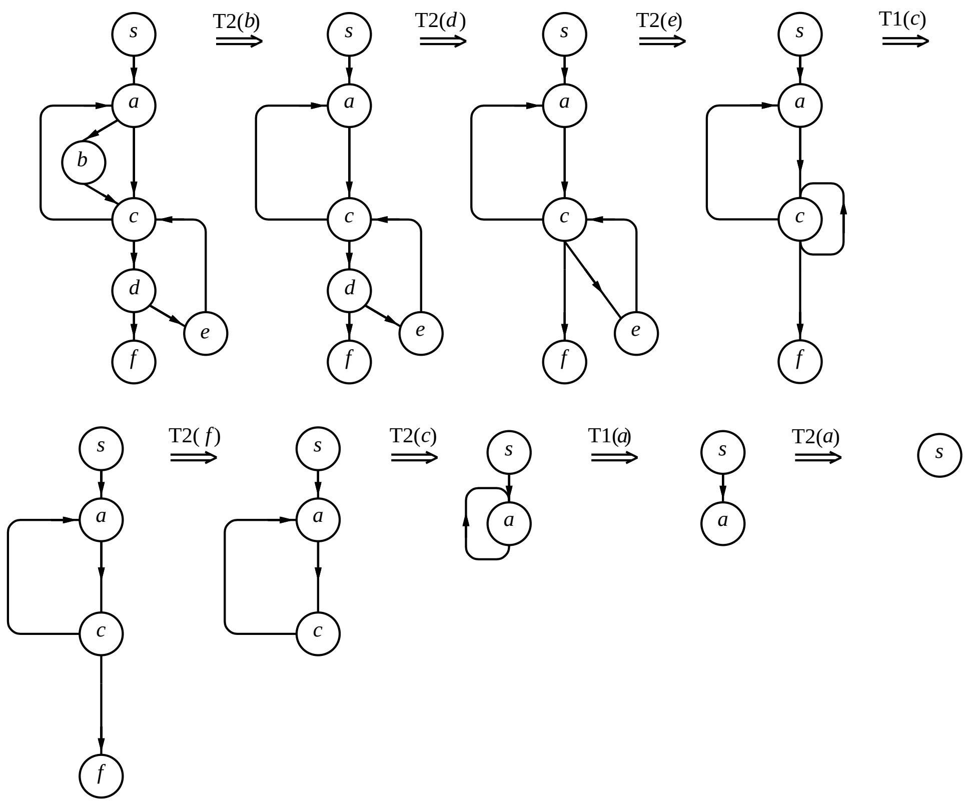


Fig. 4. An example of application of the T1 and T2 transformations.

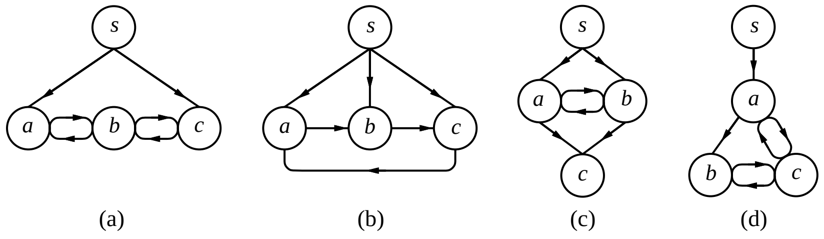


Fig. 5. Examples of extensions of the basic irreducible control flow graph of Figure 2.

shows how node splitting can be applied straightforwardly to reduce an irreducible graph.

3.1 Node Splitting

Node splitting is a technique that converts a graph G_1 into an equivalent graph G_2. We assign a label to each node of a graph; the label of node x_i is denoted $label\,(x_i)$. Duplication of a node creates a new node with the same label. An equivalence relation between two flow graphs is derived from Hecht [1977] and given in the following:

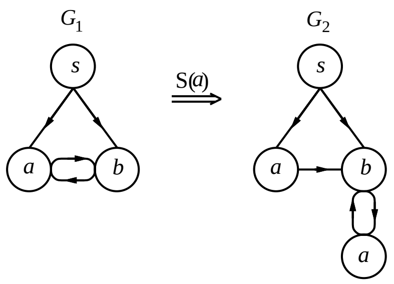


Fig. 6. A simple example of applying node splitting to node a.

Definition 3.1.1. If $P = (x_1, \ . \ . \ . \ , x_k)$ is a path in a flow graph, then define $Labels(P)$ to be a sequence of labels corresponding to this path, i.e., $Labels(P) = (label\,(x_1)\,, \ . \ . \ . \ , label\,(x_k))$. Two flow graphs G_1 and G_2 are equivalent if and only if, for each path P in G_1, there is a path Q in G_2 such that $Labels\,(P) = Labels\,(Q)$, and conversely.

According to this definition the two flow graphs of Figure 6 are equivalent. Note that all nodes a have the same $label(a)$. Node splitting is defined as follows:

Definition 3.1.2. Node splitting is a transformation of a graph $G_1 = (N, E, s)$ into a graph $G_2 = (N', E', s)$ such that a node $n \in N$, having multiple predecessors p_i, is split. For any incoming edge (p_i, n) a duplicate n_i of n is made, having only one incoming edge (p_i, n_i) and the same outgoing edges as n. N' is defined as $N' = N \cup \{n_i\} - \{n\}$ and $E' = E - \{(p_i, n)\,, (n, r_j)\} \cup \{(p_i, n_i)\,, (n_i, r_j)\}$, where r_j is a successor node of n. This transformation is denoted as $G_1 \overset{\mathrm{S}(n)}{\Rightarrow} G_2$, where $\mathrm{S}\,(n)$ is the splitting of node $n \in N$.

The principle of node splitting is illustrated in Figure 6; node a of graph G_1 is split.

THEOREM 3.1.3. *The equivalence relation between two graphs is preserved under the transformation* $G_1 \overset{S}{\Rightarrow} G_2$.

PROOF. We show that node splitting transforms any graph G_1 into an equivalent split graph G_2. Assume graph G_1 has a node v with $n > 1$ predecessors u_i and with $m \geq 0$ successors w_k, as shown in Figure 7(a). The set of $Labels(P)$ for all paths P of a graph G is denoted as $LABELS\,(G)$. With the label notation all paths of graph G_1 of Figure 7(a) are described with

$$LABELS\,(G_1) = \cup_{i=1}^{n} \cup_{k=0}^{m} \ \{(label\,(s)\,, label\,(u_i)\,, \\ label\,(v)\,, label\,(w_k))\}\,.$$

If node v is split in n copies named v_i, the split graph G_2 results. The set of all paths of graph G_2 is

$$LABELS\,(G_2) = \cup_{i=1}^{n} \cup_{k=0}^{m} \ \{(label\,(s)\,, label\,(u_i)\,, \\ label\,(v_i)\,, label\,(w_k))\}\,.$$

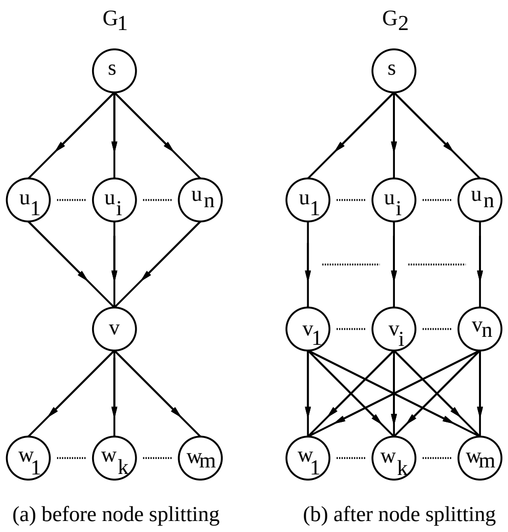


Fig. 7. Two equivalent graphs.

This graph is given in Figure 7(b). Since *label* (v_i) = *label* (v), every path in G_2 exists also in G_1, and conversely. This leads to the conclusion that the graphs G_1 and G_2 are equivalent. Since in Figure 7 we split a node with an arbitrary number of incoming and outgoing edges, we may in general conclude that splitting a node of any graph results in an equivalent graph. Using the same reasoning it will be clear that the equivalence relation is transitive. Splitting a finite number of nodes in either the original graph or any of its equivalent graphs results in a graph that is equivalent to the original graph. □

The name "node splitting" is deceptive because it suggests that the node is split in different parts, but in fact the node is duplicated.

3.2 Uncontrolled Node Splitting

The node splitting transformation technique can be used to convert an irreducible control flow graph into a reducible control flow graph. From Hecht [1977] we adopt Theorem 3.2.1.

THEOREM 3.2.1. *Let S denote the splitting of a node, and let T denote some graph reduction transformation (e.g., $T = (T1 \mid T2)^*$). Then any control flow graph can be transformed into a single node by the transformation represented by the regular expression $T(ST)^*$.*

The proof of the theorem is given in Hecht [1977].

As a result of a T2 transformation, a single node in the limit graph may correspond to multiple nodes in the original graph. Therefore, when a node is split in the limit graph, its corresponding nodes in the original graph must be split to remove irreducibility. This relation is illustrated in Figure 8. When G is transformed to a single node by a sequence of S and T transformations, the same sequence of S transformations transforms G into a reducible graph G^*. In this way the single node corresponds to all nodes of G^*.

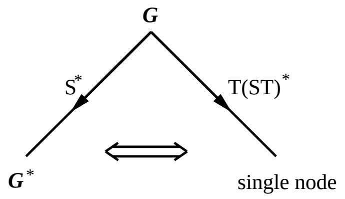


Fig. 8. Relation between the original graph and the reduced graph.

Hecht et al. describe a straightforward application of node splitting to reduce irreducible control flow graphs. This method arbitrarily selects a node for splitting from the limit graph if it has multiple predecessors. The selected node is split into several identical copies, one for each entering edge. This approach has the advantage that it is rather simple, but it has the disadvantage that it can select nodes that did not have to be split to make a graph reducible. In Figure 9(a) we see that the nodes a, b, c, and d are candidate nodes for splitting. In Figure 9(b) node d is split; the number of nodes reduces after the application of two T2 transformations, but the graph stays irreducible. Splitting of node a does not make the graph reducible; see Figure 9(c). Only splitting of node b or c converts the graph into a reducible control flow graph; see Figure 9(d). To transform the flow graph of Figure 9(d) into a single node, the following transformation sequence can be used: T2(b) T2(b') T1(c) T2(c) T2(d) T1(a) T2(a).

Although this method does inefficient node splitting, it does transform an irreducible control flow graph eventually into a reducible one. The consequence of this inefficient node splitting is that the number of duplications becomes unnecessarily large.

4. PRESENTATION OF CONTROLLED NODE SPLITTING

The problem of existing methods is that the resulting code size after converting an irreducible graph can grow uncontrolled. Controlled Node Splitting (CNS) controls the number of copies, which results in a smaller growth of the code size. CNS restricts the set of candidate nodes for splitting. First, we introduce the necessary terminology.

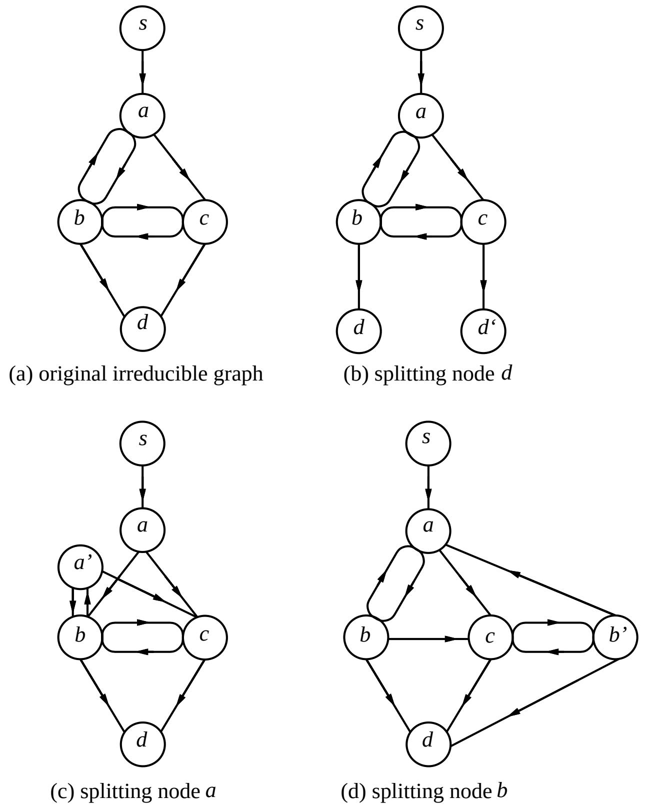
Definition 4.1. A loop in a flow graph is a path $(n_1, \ldots, n_k)$ where n_1 is an immediate successor of n_k. The nodes n_i do not have to be unique. The set of nodes contained in the loop is called a loop-set.

In Figure 9(a) $\{a, b\}$, $\{b, c\}$, and $\{a, c, b\}$ are loop-sets. The largest possible loop-set is a strongly connected component.

Definition 4.2. An immediate dominator of a node u, ID(u), is the last dominator on any path from the initial node s of a graph to u, excluding node u itself.

In Figure 1 node a dominates the nodes a, b, c, d, e, and f, but it immediately dominates only the nodes b and c.

Definition 4.3. A Shared External Dominator set (SED-set) is a subset of a loop-set L with the properties that it has only elements that share the same immediate



(a) original irreducible graph (b) splitting node *d*

(c) splitting node *a* (d) splitting node *b*

Fig. 9. Examples of node splitting.

dominator and the immediate dominator is not part of the loop-set L. A SED-set of a loop-set L is defined as

$$\text{SED-set}(L) = \{n_i \in L | \text{ID}\,(n_i) = d, d \notin L\}\,.$$

Definition 4.4. A Maximal Shared External Dominator set (MSED-set) K is defined as:

$$\text{SED-set } K \text{ is maximal} \Leftrightarrow \nexists \text{ SED-set } M,\ K \subset M \text{ and } K, M \subseteq L.$$

The definition says that an MSED-set cannot be a proper subset of another SED-set. In Figure 5(a) multiple SED-sets can be identified such as $\{a, b\}$, $\{b, c\}$, and $\{a, b, c\}$. However, there is only one MSED-set: $\{a, b, c\}$.

Definition 4.5. Nodes in an SED-set of a flow graph can be classified into three sets:

—Common Nodes (CN): Nodes that dominate other SED-set(s) and are not reachable from the SED-set(s) they dominate.

—Reachable Common nodes (RC): Nodes that dominate other SED-set(s) and are reachable from the SED-set(s) they dominate.

—Normal Nodes (NN): Nodes of an SED-set that are not classified in one of the preceding classes. These nodes dominate no other SED-sets.

In the initial graph of Figure 10(a) we can identify the MSED-sets $\{a, b\}$ and $\{c, d\}$. The nodes a, c, and d are elements of the set NN, and node b is an element of the set RC. If the edge (c, b) were not present, then node b would be an element of the set CN. Note that loop (b, c) is not an SED-set.

Theorem 4.6. *An MSED-set(L) has one node if and only if the corresponding loop L has a single header and is reducible.*

The proof of this theorem can be derived from Hecht [1977]. An example of an MSED-set that contains only one node is the graph in Figure 4 just before the transformation T1(a).

In Section 4.1 a description of CNS is given. It treats a method for minimizing the number of nodes to split. Section 4.2 gives a method for minimizing the number of copies. The number of copies is not equal to the number of splits because a split creates a copy for every entering edge. If a node has n entering edges then one split creates $n - 1$ copies. To speed up the process for minimizing the number of copies a heuristic is introduced. The algorithms implementing this heuristic are presented in Section 4.3.

4.1 Controlled Node Splitting

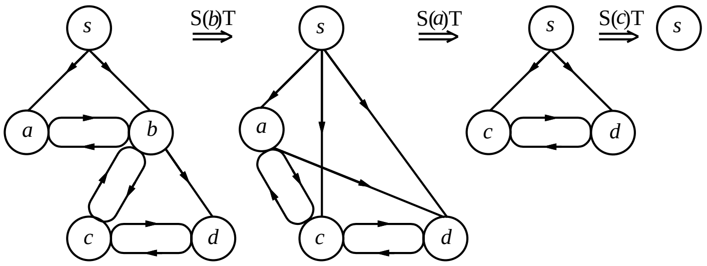
All nodes of an irreducible limit graph, except the initial node s of the graph, are possible candidates for node splitting, since they have at least two predecessors. However, as shown in Section 3.2, not all nodes are equally good candidates for splitting. CNS minimizes the number of splits. To accomplish this, two restrictions are made to the set of candidate nodes. These restrictions are as follows:

(1) Only nodes that are elements of an SED-set are candidates for splitting.
(2) Nodes that are elements of RC are not candidates for splitting.

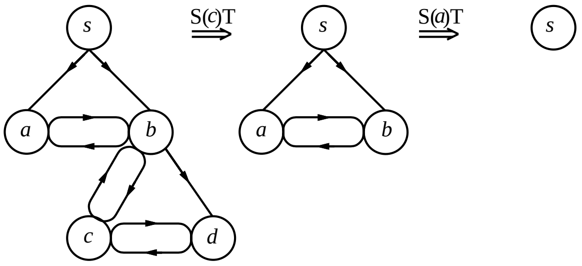
Splitting a node which is not part of an SED-set is inefficient and unnecessary, since it does not reduce the number of nodes in a loop. These nodes are automatically reduced when all SED-sets are reduced to single nodes. An example of such a split was shown in Figure 9(b) (the only SED-set in Figure 9(b) is $\{b, c\}$). The first restriction prevents such a splitting.

The second restriction is more complicated. The impact of this restriction is illustrated in Figure 10. This figure shows two different sequences of node splitting. The initial graph of the figure is a graph on which T has been applied. In Figure 10(a) there are three splits needed and in Figure 10(b) only two. In Figure 10(a) node b is split; this node, however, is an element of the set RC. Splitting node b merges the two MSED-sets $\{a, b\}$ and $\{c, d\}$. We prove later that merging MSED-sets results in more splits to reduce a graph than reducing the MSED-sets separately.

Node splitting with the preceding restrictions, alternated with T1 and T2 transformations, will eventually result in a single node. This can be seen easily. Every time a node that is an element of an SED-set is split, it is reduced by the T2 transformation and the number of nodes involved in SED-sets decreases by one.



(a) node splitting sequence of three nodes



(b) node splitting sequence of two nodes

Fig. 10. Graph with two different split graphs.

Since we are considering flow graphs with a finite number of nodes, a single node eventually remains.

THEOREM 4.1.1. *The minimum number of splits needed to reduce an MSED-set with k nodes is given by*

$$R_{splits} = k - 1.$$

PROOF. Every time a node is split and T is applied the number of nodes in the MSED-set decreases by one. For every predecessor of the node to be split, a duplicate is made; this has as a consequence that every duplicate has only one predecessor and all the duplicates can be reduced by the T2 transformation. This results in an MSED-set with one node less than the original MSED-set. To reduce the complete MSED-set, all nodes but one of the MSED-set must be split until there is only one node left. This results in $k - 1$ splits. □

THEOREM 4.1.2. *The minimum number of splits needed to convert an irreducible graph, with n MSED-sets, into a reducible graph is given by*

$$R_{total} = \sum_{i=1}^{n} (k_i - 1),$$

where R_{total} is the total number of splits, and k_i is the number of nodes of MSED-set i.

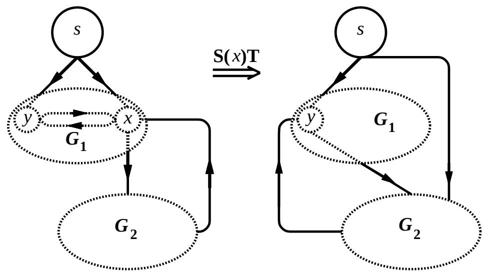


Fig. 11. Merging of two MSED-sets.

Proof. The proof consists of multiple parts; first, four related lemmas are proven.

Lemma 4.1.3. *All MSED-sets are disjoint; no two MSED-sets share a node.*

Proof. If a node is shared by two MSED-sets, then this node must have two different immediate dominators. This conflicts, however, with the definition of an immediate dominator as given in Definition 4.2. □

Since the MSED-sets are disjoint, the number of splits of the individual MSED-sets can be added. If, however, splitting nodes results in merging MSED-sets, this result does not hold any more. We have to prove that CNS does not merge MSED-sets and that merging MSED-sets does not lead to fewer splits.

Lemma 4.1.4. *Splitting a node that is part of an MSED-set and is not in RC does not result in merging MSED-sets.*

Proof. We prove that splitting a node that is an element of RC merges MSED-sets first. Afterward we prove that the splitting of nodes that are elements of CN or NN do not merge MSED-sets.

—Splitting an RC node merges two MSED-sets. Consider the graph of Figure 11. Suppose that subgraphs G_1 and G_2 are both MSED-sets and that node x is the immediate dominator of G_2. The nodes of both subgraphs form a joined loop because it is possible to go from G_1 to G_2 and vice versa. The reason that both subgraphs do not form a single MSED-set is the fact that they have different immediate dominators. By splitting a node that is in RC, in this case node x, and applying T to the complete graph, the immediate dominator of subgraph G_1 also becomes the immediate dominator of subgraph G_2. The MSED-sets are merged, since the subgraphs add up to a single loop and share the same immediate dominator. This holds also in the general case where x dominates and is reachable by n MSED-sets.

—Splitting nodes that are not in RC does not merge MSED-sets. There are now two node types left that are candidates for splitting: these are the nodes of the sets NN and CN.

  —Splitting nodes that are elements of the set NN does not merge MSED-sets. These nodes do not have edges that go to other MSED-sets; therefore splitting of these nodes does not affect the edges from one MSED-set to another, and thus the splitting will never result in merging MSED-sets.

—Splitting nodes that are elements of the set CN does not merge MSED-sets. These nodes do not form a loop with the MSED-set they dominate. By splitting such a node, the nodes of both MSED-sets get the same immediate dominator; but there is no loop between the MSED-sets, and therefore they are not merged. □

LEMMA 4.1.5. *Reducing two merged MSED-sets results in more splits to reduce a graph than reducing the MSED-sets separately.*

PROOF. Suppose MSED-set$_1$ consists of x nodes and that MSED-set$_2$ has y nodes. Merging them costs one split, since the RC node must be split. Reducing the resulting MSED-set which has now $x - 1 + y$ nodes costs $(x - 1 + y) - 1$ splits. The total number of splits is $x - 1 + y$. Reducing the two MSED-sets separately results in $(x - 1) + (y - 1)$ splits. This is one split less than the splits needed when merging the MSED-sets. □

The combination of Lemmas 4.1.4 and 4.1.5 justifies the restriction to prevent the splitting of nodes that are elements of RC.

LEMMA 4.1.6. *There always exists a node in an irreducible graph that is a member of an MSED-set, but is not an element of RC.*

PROOF. If all nodes of all MSED-sets are elements of RC, then these nodes must dominate at least two other nodes because a node cannot dominate its own dominators. These nodes are also elements of an MSED-set and of RC. The graph therefore must have an infinite number of nodes. Since we are considering graphs with a finite number of nodes, a node that is a member of an MSED-set and not an element of RC must exist. □

Since MSED-sets are disjoint, and our algorithm can always find a node that can be split without merging MSED-sets, Theorem 4.1.2 holds. □

Example 4.1.7. The MSED-sets $\{a, b\}$ and $\{c, d\}$ can be identified in Figure 10. They both have two nodes. This results in a minimal number of $(2 - 1) + (2 - 1) = 2$ splits needed to reduce the graph.

4.2 Minimizing the Number of Copies

We saw in the previous section that the algorithm minimizes the number of splits, but this does not result in a minimum number of copied instructions or basic blocks. The quantity to minimize is denoted by Q; $Q(n)$ means the quantity of node n. The quantity of a graph G is denoted by $Q(G)$ and defined as

$$Q(G) = \sum_{n \in N} Q(n).$$

The purpose of CNS is to minimize $Q(G^*)$, where G^* is the equivalent reducible representation of G. The following conditions must be satisfied to achieve this minimum:

(1) The freedom of selecting nodes to split must be as far-reaching as possible. Notice that the number of splits is also minimized if we prevent the splitting of

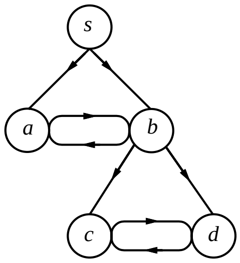


Fig. 12. A graph that has a common node that is not in the set RC.

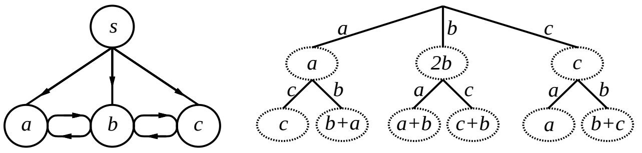
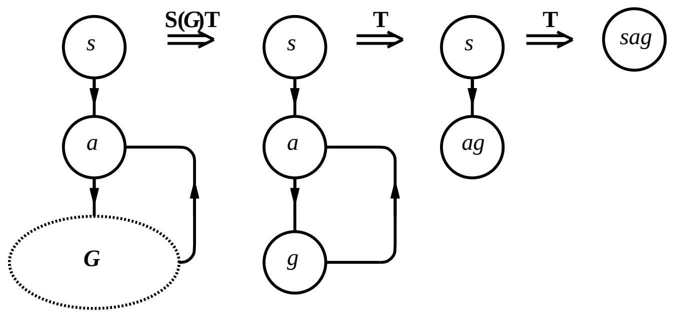


Fig. 13. An irreducible graph with its copy tree.

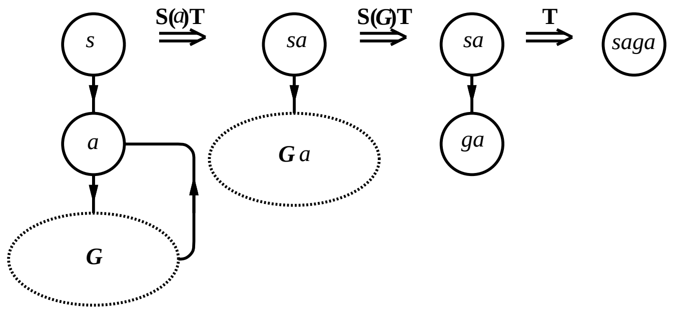
all nodes that dominate another MSED-set, i.e., prevent the splitting of nodes that are elements of RC and CN. Nevertheless, this has the disadvantage that we lose some freedom in selecting nodes. The effect of less freedom is illustrated in Figure 12. Suppose that the nodes contain a number of instructions and that we want to minimize the total resulting code size, which means that we would like to copy as few instructions as possible. The minimal number of copied instructions, if we prevent splitting nodes that are elements of RC and CN, is $Q\,(a) + \min\,(Q(c), Q\,(d))$. If we only prevent the splitting of nodes that are elements of RC, the minimal number of copied instructions is $\min\,(Q(a), Q\,(b))+ \min\,(Q(c), Q\,(d))$. If the number of instructions in node b is fewer than in node a, then the number of copied instructions is fewer in the latter case. Thus, keeping the set of candidate nodes as large as possible pays off if one would like to minimize the copied quantity.

(2) The sequence of splitting nodes must be chosen optimally. There exist multiple split sequences to solve an irreducible graph. A tree can be built to discover them all. Figure 13 shows a flow graph and the tree with all possible split sequences. The nodes of this tree indicate how many copies are introduced by the split. The edges give the split sequence. The number of copies can be found by following a path from the root to a leaf and adding the quantities of the nodes. Suppose that the nodes contain a number of instructions and that we want to minimize the total resulting code size. This means that we would like to copy as few instructions as possible. To accomplish this we can choose from six different split sequences with five different numbers of copies. The minimum number of copied instructions is $\min(Q(a + c), Q(2a + b), Q(a + 3b), Q(3b+c), Q(b+2c))$. The problem is to pick a split sequence that minimizes the number of copied instructions.

THEOREM 4.2.1. *Minimizing the $Q(G^*)$ of an irreducible graph that is converted*



(a) splitting nodes that are not in the set RC



(b) splitting a node that is in the set RC

Fig. 14. The influence on the number of copies by splitting an RC node.

to a reducible graph requires a minimum number of splits. In short,

$$Q(G^*) \textit{ is minimal } \Rightarrow \textit{ \# splits to produce } G^* \textit{ is minimal.}$$

Proof. Suppose all nodes of a limit flow graph, except the initial node s, are candidates for splitting; then nodes that are not in an MSED-set, and nodes that are elements of RC are also candidates. Splitting a node of one of these categories results in a number of splits that is greater than the minimal number of splits. If we can prove that splitting these nodes always results in a $Q(G^*)$ that is greater than the one we obtain if we exclude these nodes, then we have proven that a minimum number of splits is required in order to minimize $Q(G^*)$.

—Splitting a node that is not in an MSED-set cannot result in the minimum $Q(G^*)$. As seen in the previous section, splitting of nodes that are not in an MSED-set does not make a graph more reducible, since splitting these nodes does not decrease the number of nodes in an MSED-set. This means that the MSED-set still needs the same number of splits.

—Splitting nodes that are elements of RC cannot result in the minimum $Q(G^*)$. Consider the graph of Figure 14. In this figure the subgraph G has at least one MSED-set; otherwise the graph would not be irreducible. Figure 14(a) shows the reduction of a graph in the case where splitting of an RC node is not allowed, and Figure 14(b) shows the case where splitting of such a node is allowed. The node

g is the reduced subgraph G, and the notation sa in a node means that node s has consumed a copy of node a. The resulting quantity of the node is the sum of the quantities of nodes s and a. As we can see, the resulting total quantity of the split sequence of Figure 14(a) is $Q(s) + Q(a) + Q(g)$, and the resulting total quantity of the reduced graph of Figure 14(b) is $Q(s) + 2Q(a) + Q(g)$. Without loss of generality we can conclude that splitting a node that is in RC never can lead to the minimum total quantity. $\square$

As one can easily see, the more nodes in MSED-sets the larger the tree. This will increase the number of possible split sequences. It takes much time to compute all possibilities; therefore, a heuristic is constructed that picks a node n_i to split with the smallest $H(n_i)$ as defined by

$$H(n_i) = Q(n_i) * (\#\ predecessor\ nodes - 1).$$

The results of this heuristic, compared to the best possible split sequence, are given in Section 5.

4.3 Algorithms

The method described in the previous sections detects an irreducible control flow graph and converts it to a reducible control flow graph. In this section the algorithm for this method is given. The algorithm consists of three parts: the T1 and T2 transformations, the selection of a candidate node, and the splitting of a node.

Algorithm 4.3.1 (*Controlled Node Splitting*).

```
Input  : The control flow graph $G$ of the procedure.
Output : The reducible control flow graph $G^*$ of the procedure.
Method :
    (1)  Copy $G$ to $G'$
    (2)  Apply repeatedly T1-T2 transformations to $G'$
    (3)  **while** $G'$ has more than one node **do**
    (4)      Node selection
    (5)      Split candidate node in both $G$ and $G'$
    (6)      Apply repeatedly T1-T2 transformations to $G'$
    (7)  **endwhile**
    (8)  **return** $G^* = G$
```

Algorithm 4.3.1 expects as input a control flow graph. The structure of this flow graph is copied to a flow graph of nodes G'; see line (1). Now we have two flow graphs: a flow graph of basic blocks and a flow graph of nodes. This means that initially every node represents a basic block. Every duplicate introduced by splitting a node in G' initiates the duplication of basic blocks in node G. Due to the T2 transformation a node can correspond to multiple basic blocks; all these corresponding basic blocks must be duplicated. In line (2) the T1 and T2 transformations are applied until the graph G' does not change any more. Note that the dominator relations of the remaining nodes are not changed, and therefore no recomputing is required. If the graph of nodes is reduced to a single node, the graph is reducible, and no splitting is needed. However, if multiple nodes remain, node splitting must be applied. First, a node for splitting is selected (4). This is done with Algorithm 4.3.2, which is discussed next. The selected node is then split (5) as defined in Definition 3.1.2. In the graph of basic blocks, the corresponding basic blocks are copied also. After splitting, the T1 and T2 transformations are applied again on the graph of nodes (6). When there is still more than one node left the process starts over again. The algorithm terminates if the graph of nodes is reduced to a single node, and thus the graph of basic blocks is converted to a reducible flow graph. This graph G^* is returned. At this point the dominator relations of the flow graph of basic blocks must be recomputed, since new basic blocks are inserted.[2]

Algorithm 4.3.2 (*Node Selection*).

```
Input   : The control flow graph of nodes.
Output  : A node for splitting.
Method  :
     (1)  min = $\infty$
     (2)  **for** all nodes $n$ **do**
     (3)      **if** $n$ in an SED-set **and** $n$ not in RC **then**
     (4)          Calculate $H(n)$
     (5)          **if** $H(n) <$ min
     (6)              min = $H(n)$
     (7)              candidate node = $n$
     (8)          **endif**
     (9)      **endif**
     (10) **endfor**
     (11) **return** candidate node
```

Algorithm 4.3.2 selects a node for splitting. Initially every node is a candidate. A node is rejected if it does not fulfill the restrictions (3), as discussed in Section 4.1. For all nodes that fulfill these restrictions the heuristic $H(n)$ is calculated (4). The node that minimizes $H(n)$ is selected for splitting.

5. RESULTS

The goal of our experiments is to measure the quality of controlled node splitting in the sense of minimizing the number of copies. In the experiments four methods for node splitting are used:

—*Optimal Node Splitting* (*ONS*). This method computes the best possible node split sequence with respect to the quantity to minimize. This algorithm, however, requires a lot of computation time (up to several days on an HP735 workstation).

—*Uncontrolled Node Splitting* (*UCNS*). This is a straightforward application of node splitting; no restrictions are made to the set of nodes that are candidates for splitting. This method is described in Hecht [1977].

—*Controlled Node Splitting* (*CNS*). This method is node splitting with the restrictions discussed in Section 4.1.

[2]The algorithms for the T1 and T2 transformations and for node splitting are quite straightforward and are not given here.

Table I. The Number of Copied Basic Blocks

Procedure(Program)	**Basic Blocks**	**ONS**		**UCNS**		**CNS**		**CNSH**	
atof_generic(a68)	93	1	(1%)	33.7	(36%)	3.0	(3%)	1.0	(1%)
equals(a68)	13	1	(8%)	5.5	(42%)	5.5	(42%)	1.0	(8%)
lex(bison)	142	4	(3%)	165.3	(116%)	24.3	(17%)	4.0	(3%)
output_program(bison)	14	2	(14%)	9.7	(69%)	9.7	(69%)	2.0	(14%)
copy_definition(bison)	119	2	(2%)	417.0	(350%)	27.7	(23%)	2.0	(2%)
copy_guard(bison)	190	4	(2%)	2273.5	(1197%)	133.4	(70%)	4.0	(2%)
copy_action(bison)	183	2	(1%)	636.2	(348%)	27.7	(15%)	2.0	(1%)
next_file(expand)	17	1	(6%)	5.0	(29%)	5.0	(29%)	1.0	(6%)
re_compile_pattern(gawk)	787	1	(0%)	1202.7	(153%)	47.5	(6%)	1.0	(0%)
interp(gs)	202	5	(2%)	120.3	(60%)	93.0	(46%)	5.0	(2%)
sreadhex(gs)	33	10	(30%)	22.5	(68%)	13.0	(39%)	10.0	(30%)
gs_type1_interpret(gs)	173	4	(2%)	165.3	(96%)	165.3	(96%)	4.0	(2%)
s_LZWD_read_buf(gs)	45	14	(31%)	21.0	(47%)	21.0	(47%)	14.0	(31%)
copy_block(gzip)	17	2	(12%)	2.5	(15%)	2.5	(15%)	2.0	(12%)
compile_program(sed)	145	1	(1%)	80.1	(55%)	60.0	(41%)	1.0	(1%)
re_search_2(sed)	486	20	(4%)	1328.7	(273%)	50.0	(10%)	21.0	(4%)
squeeze_filter(tr)	33	8	(2%)	16.3	(49%)	15.5	(47%)	8.0	(24%)
Total	2692	82	(3.0%)	6505.3	(241.7%)	704.1	(26.2%)	83	(3.1%)

—*Controlled Node Splitting with Heuristic* (*CNSH*). This is the same method as CNS, but now heuristic $H(n)$ is used to select a node from the set of candidate nodes.

The algorithms are applied to a selective group of benchmarks. These benchmarks are procedures with an irreducible control flow graph and are obtained from the real-world programs: a68, bison, expand, gawk, gs, gzip, sed, tr. The programs are compiled with the GCC compiler, ported to an RISC architecture.[3] The number of copies of two different quantities is considered: in Table I the number of copied basic blocks is listed, and in Table II the number of copied instructions is listed. The reported results of the methods UCNS, CNS, and CNSH are the averages of all possible split sequences.

The first column in Tables I and II lists the procedure name, with the program name within parentheses. The second column gives the number of basic blocks or instructions of the procedure before an algorithm is applied. The other columns give the number of copies that result from the algorithms; both the absolute number of copies and a percentage that indicates the growth of the quantity with respect to the original quantity are given.

From the results of the ONS method we may conclude that node splitting does not have to lead to an excessive number of copies and that CNS outperforms UCNS. UCNS can lead to an enormous amount of copies; the average percentage of growth in basic blocks is 241.7%, and in code size it is 235.5%. CNS performs better, a growth of 26.2% for basic blocks and 30.1% for the number of instructions, but there is still a big gap with the optimal case. When using the heuristic, controlled node splitting performs very close to the optimum. The average growth in basic

[3]We used an RISC-like MOVE architecture. The MOVE project [Corporaal 1997; Hoogerbrugge and Corporaal 1994] researches the generation of application-specific processors (ASPs) by means of Transport Triggered Architectures (TTA).

Table II. The Number of Copied Instructions

Procedure(Program)	Insn	ONS	UCNS	CNS	CNSH
atof_generic(a68)	550	2 (0%)	186.2 (34%)	10.5 (2%)	2.0 (0%)
equals(a68)	84	1 (1%)	37.5 (45%)	37.5 (45%)	1.0 (1%)
lex(bison)	529	18 (3%)	577.3 (109%)	94.0 (18%)	18.0 (3%)
output_program(bison)	59	9 (15%)	41.5 (70%)	41.5 (70%)	9.0 (15%)
copy_definition(bison)	539	9 (2%)	1870.0 (347%)	122.5 (23%)	9.0 (2%)
copy_guard(bison)	880	18 (2%)	10408.2 (1183%)	603.3 (69%)	18.0 (2%)
copy_action(bison)	858	9 (1%)	2961.4 (345%)	122.5 (14%)	9.0 (1%)
next_file(expand)	64	1 (2%)	16.5 (26%)	16.5 (26%)	1.0 (2%)
re_compile_pattern(gawk)	2746	1 (0%)	4106.9 (150%)	218.5 (8%)	1.0 (0%)
interp(gs)	969	20 (2%)	588.1 (61%)	442.5 (46%)	20.0 (2%)
sreadhex(gs)	150	47 (31%)	79.7 (53%)	58.0 (39%)	47.0 (31%)
gs_type1_interpret(gs)	1175	19 (2%)	1063.8 (90%)	1063.8 (90%)	19.0 (2%)
s_LZWD_read_buf(gs)	228	62 (27%)	95.0 (42%)	95.0 (42%)	62.0 (27%)
copy_block(gzip)	88	4 (5%)	7.5 (9%)	7.5 (9%)	4.0 (5%)
compile_program(sed)	693	2 (0%)	391.4 (56%)	267.5 (39%)	2.0 (0%)
re_search_2(sed)	1857	91 (5%)	4803.7 (259%)	227.5 (12%)	93.0 (5%)
squeeze_filter(tr)	119	22 (18%)	57.0 (48%)	55.5 (47%)	22.0 (18%)
Total	11588	335 (2.9%)	27291.7 (235.5%)	3484.1 (30.1%)	337 (2.9%)

blocks for the methods CNSH and ONS is respectively 3.1% and 3.0%. The code size increase is 2.9% for both methods. Comparing the results of ONS and CNSH leads to the conclusion that CNSH performs very close to the optimum. In our experiments there was only one procedure with a very small difference.

The time it takes to transform a graph into a single node is small for UCNS, CNS, and CNSH. Usually it took only a few milliseconds on a HP735 workstation. The difference in execution time between the three methods is negligible. This was to be expected, since UCNS does no flow graph analysis but copies many more nodes than needed; CNS analyzes the flow graph, but copies fewer nodes; finally, CNSH also analyzes the flow graph and computes a heuristic, but copies only a few nodes. Apparently, analyzing the graph takes approximately the same time as splitting extra nodes. Computing the optimal split sequence (ONS) takes a lot of computation time, usually hours, because it has to check all possible split sequences to find the best solution. The flow graph of benchmark *lex* consisted, after applying T, of 13 nodes. This limit flow graph has two MSED–sets of three nodes each; this results ideally in four splits. However, ONS in the worst case needs 12! ($\approx 10^8$) split/transformation sequences to find the best solution.

The results in Tables I and II show a substantial improvement when using CNSH. Nevertheless, the question is what is the impact when using a simpler method such as UCNS on the code size of the complete program. If the impact is small then why bother, except for the theoretical aspects? In Tables III and IV, the effects for the complete code expansion are shown. All procedures of benchmarks that have an irreducible control flow graph are converted to procedures with a reducible control flow graph. In Table III we show the impact in basic blocks and in Table IV the impact on the code size. The first column of both tables lists the program name; the second column lists the total number of basic blocks or instructions; the remaining columns list the increase in basic blocks or in instructions for each method.

As can be seen from the tables, the impact of node splitting can be substantial in terms of the number of basic blocks or instructions. For UCNS the average increase

Table III. The Increase of Basic Blocks per Program

Program	**Basic Blocks**	**ONS**		**UCNS**		**CNS**		**CNSH**	
a68	3972	2	(0%)	39.2	(1%)	8.5	(0%)	2.0	(0%)
bison	4441	14	(0%)	3501.7	(79%)	222.8	(5%)	14.0	(0%)
expand	1226	1	(0%)	5.0	(0%)	5.0	(0%)	1.0	(0%)
gawk	8342	13	(0%)	1252.4	(15%)	63.5	(1%)	13.0	(0%)
gs	16514	47	(0%)	405.3	(2%)	326.3	(2%)	47.0	(0%)
gzip	3244	2	(0%)	2.5	(0%)	2.5	(0%)	2.0	(0%)
sed	3823	21	(1%)	1408.8	(37%)	110.0	(3%)	22.0	(1%)
tr	1554	8	(1%)	16.3	(1%)	15.5	(1%)	8.0	(1%)
Total	43116	108 (0.3%)		6631.2 (15.4%)		754.1 (1.7%)		109.0 (0.3%)	

Table IV. The Increase of Instructions per Program

Program	**Insn**	**ONS**		**UCNS**		**CNS**		**CNSH**	
a68	19527	3	(0%)	223.7	(1%)	48.0	(0%)	3.0	(0%)
bison	19689	63	(0%)	15858.4	(80%)	983.8	(5%)	63.0	(0%)
expand	4949	1	(0%)	16.5	(0%)	16.5	(0%)	1.0	(0%)
gawk	36445	56	(0%)	4356.1	(12%)	285.0	(1%)	56.0	(0%)
gs	85824	210	(0%)	2169.7	(3%)	1804.1	(2%)	210.0	(0%)
gzip	14620	4	(0%)	7.5	(0%)	7.5	(0%)	4.0	(0%)
sed	17489	93	(1%)	5195.1	(30%)	495.0	(3%)	95.0	(1%)
tr	6530	22	(0%)	57.0	(1%)	55.5	(1%)	22.0	(0%)
Total	205073	452 (0.2%)		27884.0 (13.6%)		3695.4 (1.8%)		454.0 (0.2%)	

in basic blocks is 15.4%, and in instructions it is 13.6%. For the program *bison* the code size even increases 80%. When using controlled node splitting the increases are smaller and quite acceptable. CNSH results as expected in the smallest increases for both quantities. These results show the importance of a clever transformation of irreducible control flow graphs.

6. CONCLUSIONS

A method is presented that transforms an irreducible control flow graph to a reducible control flow graph. The method is based on node splitting. To achieve the minimum number of splits the set of possible candidate nodes is limited to nodes with specific properties. Since splitting of these nodes can result in a minimum resulting code size, the algorithm can be used to prevent uncontrolled growth of the code size. Because the computation time to determine the optimum split sequence is large, a heuristic has been developed.

The method with the heuristic is called controlled node splitting with heuristic. This method is applied to a set of procedures that contain irreducible control flow graphs. The results are compared with the results of the other methods; these methods are uncontrolled node splitting and controlled node splitting. From our experiments it follows that uncontrolled node splitting can lead to an enormous number of copies; the average growth in code size per procedure is 235.5%. Controlled node splitting performs better (30.1%), but there is still a big gap with the optimal case. We observed that the average number of copies when using controlled node splitting with heuristic is very close to that of the optimum; the average growth in code size per procedure is only 2.9%.

We also looked at the impact on the total code size of the benchmarks containing procedures with irreducible control flow graphs. The same methods as for the analysis per procedure were used. For CNSH the impact on the total code size is very small, only 0.2% on average. The impact of UCNS is, however, surprisingly large: an average code size growth of 13.6% with a maximum for *bison* of 80%.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their conscientious reviews and constructive criticism, which improved the presentation of the article.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1988. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, Mass.

Allen, F. 1972. A basis for program optimization. In *Proceedings of 1971 IFIP Congress.* IEEE, New York, 385–390.

Allen, F. and Cocke, J. 1976. A program data flow analysis procedure. *Commun. ACM 19,* 3 (Mar.), 137–147.

Ammarguellat, Z. 1992. A control-flow normalization algorithm and its complexity. *IEEE Trans. Softw. Eng. 18,* 3 (Mar.), 237–251.

Bernstein, D. and Rodeh, M. 1991. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.* ACM, New York, 241–255.

Cocke, J. 1971. On certain graph-theoretic properties of programs. Tech. Rep. Res. Rep. RC-3391, T.J. Watson Research Center, Yorktown Heights, N.Y.

Cocke, J. and Miller, R. E. 1969. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Hawaii Conference on System Sciences.* IEEE, New York, 143–146.

Corporaal, H. 1997. *Microprocessor Architectures; from VLIW to TTA.* John Wiley, New York.

Dhamdhere, D. and Patil, H. 1993. An elimination algorithm for bidirectional data flow problems using edge placement. *ACM Trans. Program. Lang. Syst. 15,* 2 (Apr.), 312–336.

Erosa, A. M. and Hendren, L. J. 1994. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages.* IEEE, New York, 229–240.

Graham, S. and Wegman, M. 1976. A fast and usually linear algorithm for global data flow analysis. *J. ACM 23,* 1 (Jan.), 172–202.

Hecht, M. and Ullman, J. 1972. Flow graph reducibility. *SIAM J. Comput. 1,* 2, 188–202.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs.* Programming Languages Series. Elsevier North-Holland, Amsterdam.

Hoogerbrugge, J. and Corporaal, H. 1994. Transport-triggering vs. operation-triggering. In *Proceedings of the 5th Conference on Compiler Construction.* Lecture Notes in Computer Science, vol. 786. Springer-Verlag, Berlin, 435–449.

Ryder, B. G. and Paull, M. C. 1986. Elimination algorithms for data flow analysis. *ACM Comput. Surv. 18,* 3 (Sept.), 277–316.

Sreedhar, V. C., Gao, G. R., and Lee, Y.-F. 1996a. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst. 18,* 6 (Nov.), 649–658.

Sreedhar, V. C., Gao, G. R., and Lee, Y.-F. 1996b. A new framework for exhaustive and incremental data flow analysis using DJ graphs. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation.* ACM, New York, 278–290.

Tarjan, R. 1981. Fast algorithms for solving path problems. *J. ACM 28,* 3 (July), 594–614.

Received February 1997; accepted June 1997