Controlled Node Splitting

Johan Janssen Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {johan,heco}@duteca.et.tudelft.nl

Abstract

To exploit instruction level parallelism in programs over multiple basic blocks, programs should have reducible control flow graphs. However not all programs satisfy this property. A new method, called Controlled Node Splitting (CNS), for transforming irreducible control flow graphs to reducible control flow graphs 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 may result in an average code size increase of 250%. CNS with the heuristic limits the increase to only 3%.

Keywords: control flow graphs, reducibility, irreducibility, node splitting, compilation, instruction level parallelism.

1 Introduction

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 ILP of the target architecture as efficient as possible. This mapping is used for Very Long Instruction Word (VLIW) and superscalar architectures. The latter are used in most workstations. These architectures have instructions with multiple operations that are issued simultaneously. It is the responsibility of the compiler to fill these instructions as efficiently as possible with operations. This process is called scheduling.

Problem statement: 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 leaves always at the end. Several scheduling scopes can be found which go beyond the basic block level [1]. The most general scope currently used is called a *region* [2]. This 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 [2]. In [1] a speedup over 40% is reported using region scheduling. 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.

Exploiting ILP 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 directly be applied 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, that is an essential part of any compiler, can be done more efficiently [3].

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 [4] and [5] 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 take the resulting code size into account. Solutions for solving the problem at the back-end are given in [6, 7, 8, 9]. The solution given by Cocke and Miller [6, 9] is very time complex and it does not try to minimize the resulting code size. The method described by Hecht *et al.* [7, 8] is even more inefficient in the sense of minimizing the code size, but it requires less analysis. In this paper 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.

Paper overview: In section 2 reducible and irreducible 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, 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 we developed a heuristic 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 Flow Graphs

The control flow 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. From the initial node $s \in N$ there is a path 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 finding sufficient ILP requires as input a reducible flow graph. Many definitions for reducible flow graphs are proposed. The one we adopt is given in [8] and is based on the partitioning of the edges of a control flow graph G into two disjoint sets:

1. The set of *back edges* BE consist of all edges whose heads dominate their tails.
2. The set of *forward edges* FE consists of all edges which are not back edges, thus $FE = E - BE$.

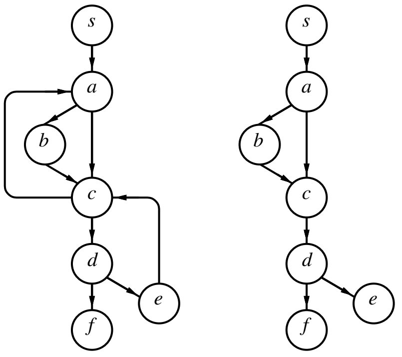


Figure 1: a) A reducible control flow graph. b) The graph $G = (N, FE, s)$.

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 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 $BE = \{(c, a), (e, c)\}$ and $FE = \{(s, a), (a, b), (a, c), (b, c), (c, d), (d, e), (d, f)\}$. The definition of a reducible flow graph is:

Definition 2.2 *A flow graph G is reducible if and only if the flow graph $G = (N, FE, s)$ 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 however 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.

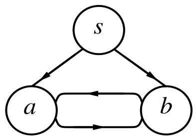


Figure 2: The basic irreducible control flow graph.

From definition 2.2 we can derive that if a control flow graph G is irreducible then 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. One of them is to use interval analysis [10, 11]. The method used here is the Hecht-Ullman T1-T2 analysis [12, 3]. This method is based on two transformations T1 and T2. These transformations are illustrated in figure 3 and are defined as:

Definition 2.3 *Let $G = (N, E, s)$ be a control flow graph and let $u \in N$. The T1 transformation removes the edge*

$(u, u) \in E$, *which is a self-loop, if this edge exists. The derived graph becomes* $G' = T1(G) = (N, E - \{(u, u)\}, s)$. *In short* $G \overset{T1(u)}{\Rightarrow} G'$.

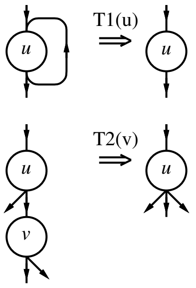


Figure 3: The T1 and T2 transformation.

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 T2 transformation 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* $G' = T2(G) = (N - \{v\}, (E - \{(v, n) \mid n \in I\} - \{(u, v)\}) \cup \{(u, n) \mid n \in I\}, s)$. *In short* $G \overset{T2(v)}{\Rightarrow} G'$.

Definition 2.5 *The graph that results when 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.*

In [7] 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 after repeatedly applying T1 and T2 transformations in any particular order the flow graph can be reduced into a single node.*

The proof of this theorem can be found in [12]. 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. Examples of irreducible graphs are given in figure 5. From theorem 2.6 it follows that we can alternatively define irreducibility by:

Corollary 2.7 *A flow graph is irreducible if and only if the limit flow graph is not a single node*[1].

[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 [12].

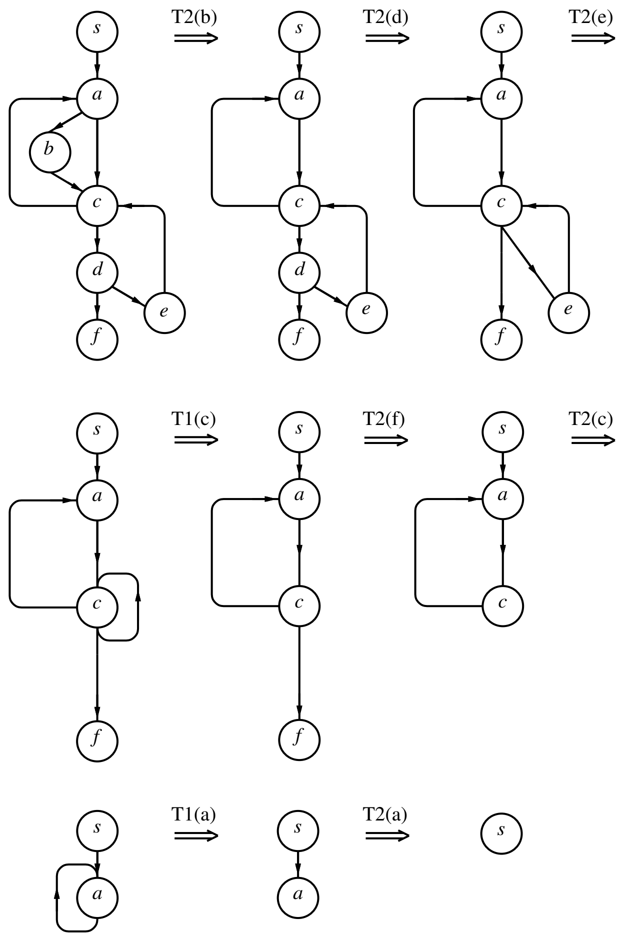


Figure 4: An example of application of the T1 and T2 transformations.

3 Flow Graph Transformation

If a control flow graph occurs to be irreducible, a graph transformation technique can be used to obtain a reducible control flow graph. In the past some methods are given to solve this problem [6, 7, 8]. Most methods for converting an irreducible control flow graph are based on a technique called node splitting. In section 3.1 this technique to reduce an irreducible flow graph is described. Section 3.2 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 to 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 [7] and given below.

Definition 3.1 *If* $P = (x_1, \ldots, x_k)$ *is a path in a flow graph, then define* $Labels(P)$ *to be a sequence of labels corresponding to this path; that is,* $Labels(P) =$

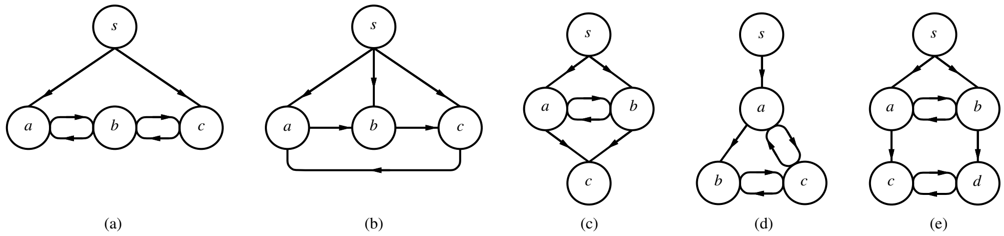


Figure 5: Examples of extensions of the basic irreducible control flow graph of figure 2.

$(label\,(x_1)\,,\ldots,label\,(x_k))$. *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. Node splitting is defined as:

Definition 3.2 *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 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* $G_1 \overset{S(n)}{\Rightarrow} G_2$*, where* $S\,(n)$ *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.

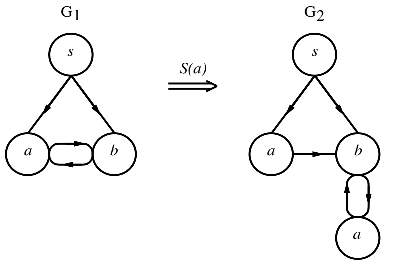


Figure 6: A simple example of applying node splitting.

Theorem 3.3 *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 $u_{\rm i}$ and with m ≥ 0 successors $w_{\rm k}$, as shown in figure 7a. The set of $Labels(P)$ for all paths P of a graph G is denoted with $LABELS\,(G)$. With the label notation all paths of graph G_1 of figure 7a are described with:

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

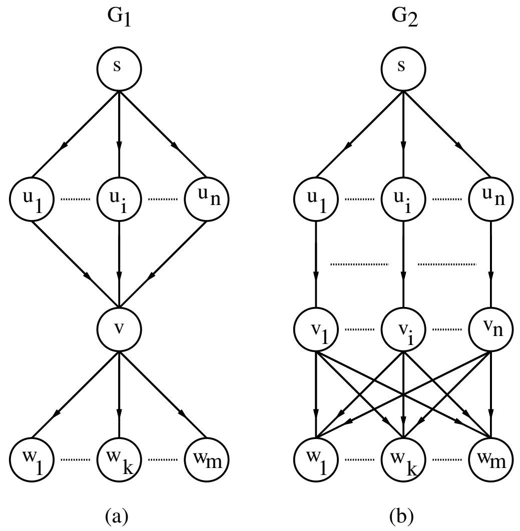


Figure 7: Two equivalent graphs, (a) before node splitting, (b) after node splitting.

If node v is split in n copies named $v_{\rm i}$ the split graph G_2 results. The set of all paths of graph G_2 is:

$$LABELS\,(G_2) = \cup_{i=1}^{n} \cup_{k=0}^{m} \quad \{(label\,(s)\,, label\,(u_i)\,, label\,(v_i)\,, label\,(w_k))\} \quad (2)$$

This graph is given in figure 7b. Since $label\,(v_{\rm i}) = label\,(v)$ 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. Without loss of generality we can 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 which 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.

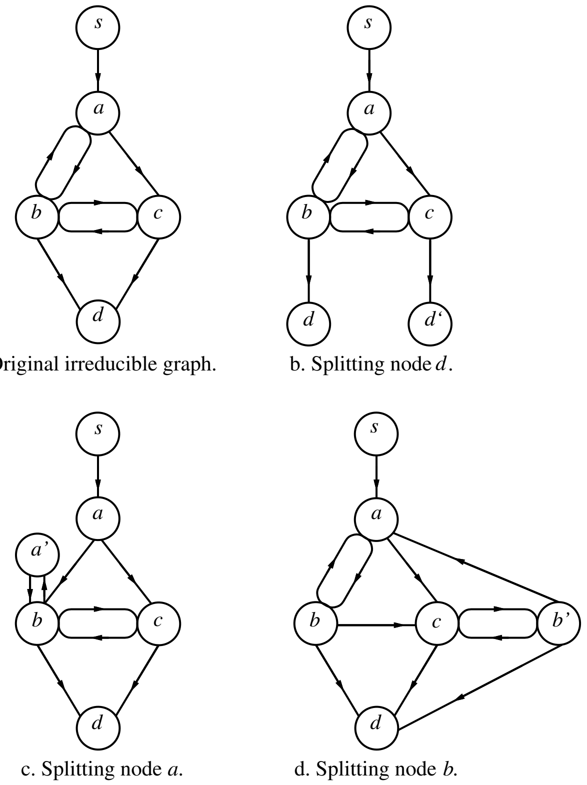
3.2 Uncontrolled Node Splitting

The transformation technique node splitting can be used to convert an irreducible control flow graph into a reducible control flow graph. From Hecht [7] we adopt theorem 3.4.

Theorem 3.4 *Let S denote the splitting of a node, and let T denote some graph reduction transformation (e.g. $T = (T1^* \, T2^*)^*$). 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 [7].

Hecht *et al.* describe a straightforward application of node splitting to reduce irreducible control flow graphs. This method selects a node for splitting from the limit graph if the node 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 8a we see that the nodes a, b, c and d are candidate nodes for splitting. In figure 8b node d is split, the number of nodes reduces after the application of two T2 transformations, but the graph is still irreducible. Splitting of node a neither makes the graph reducible, see figure 8c. Only splitting of node b or c converts the graph into a reducible control flow graph, see figure 8d.



a. Original irreducible graph. b. Splitting node d. c. Splitting node a. d. Splitting node b.

Figure 8: Examples of node splitting.

Although this method does inefficient node splitting, it does transform an irreducible control flow graph eventually in 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 amount 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 set of nodes contained in the loop is called a loop-set.*

In figure 8a $\{a, b\}$, $\{b, c\}$ and $\{a, b, c\}$ are loop-sets.

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

$$\textit{SED-set(L)} = \{n_i \in L | ID\,(n_i) = d, d \notin L\} \tag{3}$$

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

$$\textit{SED-set K is maximal} \Leftrightarrow \nexists \textit{ SED-set M, } K \subset M$$

The definition says that an MSED-set cannot be a proper subset of another SED-set. In figure 5a multiple SED-sets can be identified like $\{a, b\}$, $\{b, c\}$ and $\{a, b, c\}$. But 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 above classes. These nodes dominate no other SED-sets.*

In the initial graph of figure 9a 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) was not present then node b would be an element of the set CN. Note that loop (b, c) is not a SED-set.

Theorem 4.6 *A SED-set(L) has one node* $\Leftrightarrow$ *The corresponding loop L has a single header and is reducible.*

The proof of this theorem can be derived from [7].

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 amount of copies. The number of copies is not equal to the number of splits because a split creates for every entering edge a copy. If a node has n entering edges then one split creates n-1 copies. To speed up the process for minimizing the amount of copies a heuristic is given. The algorithms implementing these methods are given in section 4.3.

4.1 Controlled Node Splitting

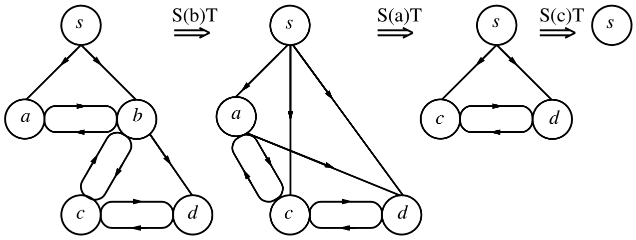
All nodes of an irreducible limit graph, except the initial node s of the graph, are possible candidates for node splitting since they have at least two predecessors. However splitting of some nodes is not efficient; see section 3.2. CNS minimizes the number of splits. To accomplish this, two restrictions are made to the set of candidate nodes. These restrictions are:

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.

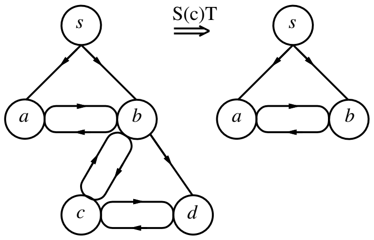
The first restriction prevents the splitting of nodes that are not in an SED-set. Splitting such a node is inefficient and unnecessary. An example of such a split was shown in figure figure 8b.

The second restriction is more complicated. The impact of this restriction is illustrated in figure 9. 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 9a there are three splits needed and in figure 9b only two. In figure 9a node b is split, this node however is an element of the set RC. The second restriction prevents a splitting sequence as the one in figure 9a.

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



a. Node splitting sequence of three nodes.



b. Node splitting sequence of two nodes.

Figure 9: Graph with two different split graphs.

Theorem 4.7 *The minimum number of splits needed to reduce an MSED-set with k nodes is given by:*

$$T_{splits} = k - 1. \tag{4}$$

Proof: Every time a node is split and T is applied, the number of nodes in the MSED-set decreases with one. For every predecessor of the node to split a duplicate is made, this means 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 4.8 *The minimum number of splits needed to convert an irreducible graph, with n MSED-sets, into a reducible graph is given by:*

$$T_{splits} = \sum_{i=1}^{n} (k_i - 1) \tag{5}$$

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

Proof: The proof consists of multiple parts, first some related lemmas are proven.

> **Lemma 4.9** *All MSED-sets are disjoint, that is there are no two MSED-sets that 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 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 anymore. Therefore we have to proof that CNS does not merge MSED-sets and that merging MSED-sets does not lead to less splits.

Lemma 4.10 *Splitting a node that is part of an MSED-set and is not in RC does not result in merging MSED-sets.*

Proof: First we shall proof that splitting a node that is an element of RC merges MSED-sets. Afterwards we proof that splitting of nodes that are elements of CN or NN do not merge MSED-sets.

– Splitting a RC node merges two MSED-sets. Consider the graph of figure 10. Suppose that subgraphs G_1 and G_2 are both MSED-sets. 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 becomes also the immediate dominator of subgraph G_2. Since the subgraphs add up to a single loop and share the same immediate dominator the MSED-sets are merged. This holds also in the case x dominates and is reachable by n MSED-sets.

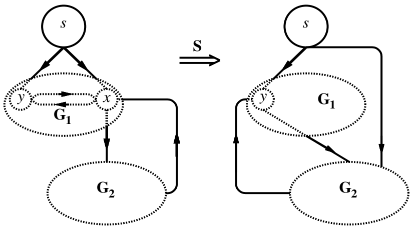


Figure 10: Merging of two MSED-sets.

– Splitting nodes that are not in RC do not merge MSED-sets. There are now two types of nodes left that are candidates for splitting, these are the nodes of the sets NN and NC.
  * Splitting nodes that are element of the set NN do 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 therefore the splitting will never result in merging MSED-sets.
  * Splitting nodes that are element of the set NC do not merge MSED-sets. These nodes do not form a loop with the MSED-set it dominates. By splitting such a node the nodes of both MSED-sets get the same immediate dominator but since the nodes do not form a loop the MSED-sets cannot be merged.

□

Lemma 4.11 *Reducing two merged MSED-sets results in more splits to reduce a graph than reducing the MSED-sets separately.*

Proof: Suppose SED-set$_1$ consists of x nodes and SED-set$_2$ consists of y nodes. Merging them costs one split since the RC node must be split. Reducing the resulting SED-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 SED-sets separately results in (x-1)+(y-1) splits. This is one split less than the splits needed when merging the SED-sets. □

The combination of lemmas 4.10 and 4.11 justifies the restriction to prevent the splitting of nodes that are elements of RC.

Lemma 4.12 *There exists always a node in a graph that is part of an MSED-set but that 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. 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 there must be a node that is part of an MSED-set but that is not an element of RC. □

Since MSED-sets are disjoint and our algorithm can always find a node that can be split without merging MSED-sets the result of equation 5 holds. □

Example 4.13 *In figure 9 the MSED-sets* $\{a, b\}$ *and* $\{c, d\}$ *can be identified. They have both two nodes. This results in a minimal number of (2-1)+(2-1) = 2 splits needed to reduce the graph.*

4.2 Minimizing the amount of copies

In the previous section we saw that the algorithm minimizes the number of splits, but this does not result in a minimum number of copies. Two conditions must be satisfied to achieve this minimum:

1. The freedom of selecting nodes to split must be as big as possible. Notice that the number of splits is also minimized if we prevent the splitting of all nodes that dominate another MSED-set, that is prevent splitting of nodes

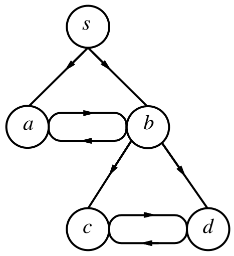
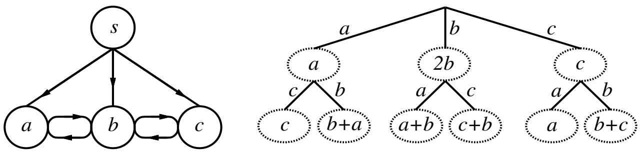


Figure 11: A graph that has common a node that is not in the set RC.

that are elements of RC and CN. But this has the disadvantage that we lose some freedom in selecting nodes. This loss of freedom is illustrated in figure 11. 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 less instructions as possible. The number of copied instructions if we prevent splitting nodes that are elements of RC and CN is: $a + \min(c, d)$. If we only prevent the splitting of nodes that are element of RC the number of copied instructions is: $\min(a, b) + \min(c, d)$. If the number of instructions in node b is less than in node a then the number of copied instructions is less in the latter case. Thus keeping the set of candidate nodes as big as possible pays off if one would like to minimize the amount of copies.

2. The sequence of splitting nodes must be chosen optimal. There exists multiple split sequences to solve an irreducible graph. A tree can be build to discover them all. The tree with all possible split sequences for the flow graph of figure 12a is drawn in figure 12b. The



a. An irreducible flow graph. b. Copy tree of figure 12a.

Figure 12: An irreducible graph with its copy tree.

nodes of the tree indicates 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, which means that we would like to copy as less instructions as possible, then we can choose from 6 different split sequences with 5 different numbers of copies. The minimum number of copied instructions is: $\min(a + c, 2a + b, a + 3b, 3b + c, b + 2c)$. The problem is to pick a split sequence that minimizes the number of copied instructions.

The quantity chosen to minimize in the above is the number of instructions, but it can also be interesting to minimize another quantity, for example the number of basic blocks. In the following the quantity to minimize is denoted with Q, $Q\ (n)$ means the quantity of node n, $Q(G)$ is the quantity of a graph G and is defined as:

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

The purpose of CNS is to minimize $Q(G^*)$, where G^* is the transformation of G into a single node using some sequence of splits.

Theorem 4.14 *Minimizing the resulting $Q(G^*)$ of an irreducible graph that is converted to a reducible graph requires a minimum number of splits, where G^* is a single node; that is the totally reduced graph. In short:*

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

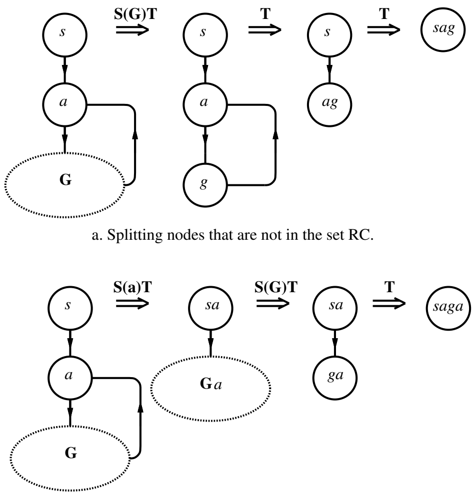
Proof: Suppose all nodes of a limit flow graph, except the initial node s, are candidates for splitting, then nodes that are not in an MSED-set and nodes that are elements of RC are also candidates. Splitting a node of one of these categories results in a number of splits that is greater than the minimal number of splits. If we can proof that splitting these nodes always result 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 do 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 element of RC cannot result in the minimum $Q(G^*)$.

  Consider the graph of figure 13. In this figure the subgraph G has at least one MSED-set, otherwise the graph would not be irreducible. Figure 13a shows the reduction of a graph in the case that splitting of a RC node is not allowed and in 13b 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 13a is Q(s)+Q(a)+Q(g) and the resulting total quantity of the reduced graph of figure 13b 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. □



a. Splitting nodes that are not in the set RC.

b. Splitting a node that is in the set RC.

Figure 13: The influence on the number of copies by splitting a RC node.

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

$$H\,(n_i) = Q(n_i) * (\#\ predecessor\ nodes - 1) \qquad (6)$$

for every candidate node. 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 (1) the T1 and T2 transformations, (2) the selection of a candidate node, and the (3) splitting of a node.

Algorithm 4.1 expects as input a control flow graph of basic blocks. The structure of this flow graph is copied to a flow graph of nodes(1). Now we have two different 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 the flow graphs of nodes is also duplicated in the flow graph of basic blocks. After the graph is copied the T1 and T2 transformations are applied till the graph of nodes does not change any more(2). If the graph of nodes is reduced to a single node, the graph is reducible and no splitting is needed. However if there remain multiple nodes, node splitting must be applied. First a node for splitting is selected(4). This is done with algorithm 4.2 that is discussed later. The selected node is then split (5) as defined in definition 3.2. In the graph of basic blocks, the corresponding basic blocks are copied also. After splitting, T1 and T2

Algorithm 4.1 *Controlled Node Splitting*

Input : *The control flow graph of a procedure.*
Output : *The reducible control flow graph of the procedure.*
Method :

```
(1)  Copy the flow graph of basic blocks
     to a flow graph G of nodes
(2)  Apply repeatedly T1-T2
     transformations to G
(3)  while G has more than one node do
(4)        Node selection
(5)        Split candidate node
(6)        Apply repeatedly T1-T2
           transformations to G
(7)  endwhile
```

transformations are applied again on the graph of nodes (6). When there is still more than one node left the process start 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.

Algorithm 4.2 *Node selection*

Input : *The control flow graph of nodes*
Output : *A node for splitting*
Method :

```
(1)  min = infinite
(2)  for all nodes n do
(3)        if n in an SED-set
           and n not in RC then
(4)              Calculate value 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
```

The algorithms for the T1 and T2 transformations and for node splitting are quite straightforward and not given here. Algorithm 4.2 selects a node for splitting. Initially every node is a candidate. A node is rejected as a candidate if it does not fulfill the restrictions (3) as discussed in subsection 4.1. For all nodes that fulfill these restrictions the heuristic is calculated (4) with equation 6. The node with the smallest heuristic 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 amount 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; the number of basic blocks or the number of instructions. This algorithm however requires a lot of computation time (up to several days on a HP735 workstation).

- Uncontrolled Node Splitting, *UCNS*. A straightforward application of node splitting, no restrictions are made to the set of nodes that are candidate for splitting.

- Controlled Node Splitting, *CNS*. Node splitting with the restrictions discussed in section 4.1.

- Controlled Node Splitting with Heuristic, *CNSH*. The same method as CNS but now a heuristic 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 which is ported to a RISC architecture[2]. The amount of copies of two different quantities are considered. In table 1 the number of copies of basic blocks are listed and in table 2 the number of copied instructions. The reported results of the methods UCNS, CNS and CNSH are the averages of all possible split sequences.

The first column in the tables 1 and 2 lists the procedure name, with the program name in 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. The absolute number of copies is given and a percentage that indicates the growth of the quantity with respect to the original quantity is given.

From the results of the ONS method we can conclude that node splitting does not have to lead to an excessive number of copies. Furthermore we can conclude 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 251.9%. CNS performs better, a growth of 26.2% for basic blocks and 32.7% for the number of instruction, 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 blocks for the methods CNSH and ONS is respectively 3.1% and 3.0% . The growth in code size is for both methods 3.1%. Comparing the results of ONS and CNSH lead to the conclusion that CNSH performs very close to the optimum. In our experiments there was only one procedure with a very small difference.

[2]We used a RISC like MOVE architecture. The MOVE project [13, 1] researches for the generation of application specific processors (ASPs) by means of Transport Triggered Architectures (TTA).

6 Conclusions

A method has been given which transforms an irreducible control flow graph to a reducible control flow graph. This gives us the opportunity to exploit ILP over a larger scope than a single basic block for any program. 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 (very) large, a heuristic has been developed.

The method with the heuristic is called controlled node splitting with heuristic. This method is compared with 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 is 251.9%. Controlled node splitting performs better (32.7%) 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 for both methods is 3.1%.

References

[1] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Lecture Notes in Computer Science 786, Compiler Construction*, pages 435–449. Springer-Verlag, 1994.

[2] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[3] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[4] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. Technical Report 76, McGill University, School of Computer Science, September 1993.

[5] Zahira Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transaction on software engineering*, 18(3):237–251, March 1992.

[6] John Cocke and Raymond E. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of 2nd Hawaii Conference on System Sciences*, pages 143–146, 1969.

[7] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Programming Languages Series. Elsevier North-Holland, 1977.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1988.

Table 1: 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%)	1328.8	(273%)	50.0	(10%)	21.0	(4%)
squeeze_filter(tr)	33	8	(2%)	16.3	(49%)	15.5	(47%)	8.0	(2%)
total	2692	82	(3.0%)	6505.4	(241.7%)	702.1	(26.2%)	83	(3.1%)

[9] J. Cocke. On certain graph-theoretic properties of programs. Technical Report Research Report RC-3391, T.J. Watson Research Center, 1971.

[10] F.E. Allen. A basis for program optimization. In *Proceedings of 1971 IFIP Congres*, pages 385–390, Amsterdam, 1971. IEEE, North Holland Publ.

[11] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–147, March 1976.

[12] M.S. Hecht and J.D. Ullman. Flow graph reducibility. *SIAM J. Computing*, 1(2):188–202, 1972.

[13] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

Table 2: The number of copied instructions.

Procedure(Program)	# instructions	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%)	4107.0	(150%)	218.5	(8%)	1.0	(0%)
interp(gs)	994	20	(2%)	600.6	(60%)	455.0	(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%)	1063.3	(90%)	1063.3	(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%)	4803.8	(259%)	227.5	(12%)	93.0	(5%)
squeeze_filter(tr)	119	22	(18%)	57.0	(48%)	55.5	(47%)	22.0	(18%)
total	10840	335	(3.1%)	27303.9	(251.9%)	3546.1	(32.7%)	337	(3.1%)