Controlled Node Splitting

Johan Janssen and 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

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 235%. 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 issue multiple instructions (or operations) simultaneously. It is the responsibility of the compiler to order these instructions as efficiently as possible. 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 instead of basic block 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 Sect. 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 Sect. 3. Our approach, Controlled Node Splitting (CNS), is described in Sect. 4. All known conversion methods convert irreducible flow graphs without minimizing the number of copies. With CNS 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.

The results of applying CNS to several benchmarks are given in Sect. 5. Finally the conclusions are given in Sect. 6.

2 Irreducible Flow Graphs

The control flow of a program can be described with a control flow graph. Its nodes represent sequences of operations or basic blocks, and its edges represent the flow of control.

Definition 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 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.

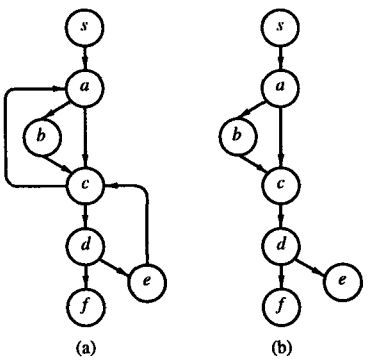


Fig. 1. a) A reducible control flow graph, b) the graph $G = (N, FE, s)$

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 $FE = E - BE$.

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

The control flow graph of Fig. 1 is reducible since $G = (N, FE, s)$ is acyclic. The control flow graph of Fig. 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.

From Definition 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 Fig. 3 and are defined as:

Definition 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 $G \overset{T1(u)}{\Rightarrow} G'$.

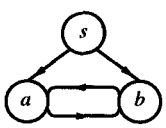


Fig.2. The basic irreducible flow graph

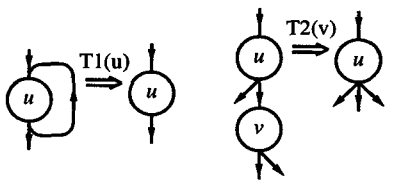


Fig.3. The T1 and T2 transformation

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

Definition 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 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 Fig. 4. The flow graph from Fig. 1 is reduced to a single node, so we can conclude that this flow graph is reducible.

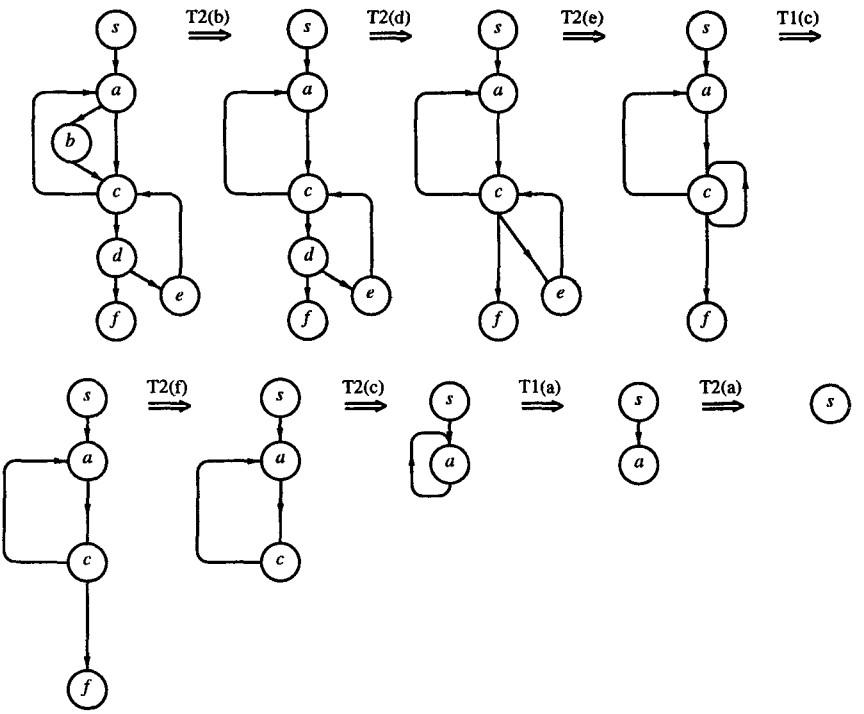


Fig. 4. An example of application of the T1 and T2 transformations

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 causes the irreducibility. Examples of irreducible graphs are given in Fig. 5. From Theorem 6 it follows that we can alternatively define irreducibility by:

Corollary 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].

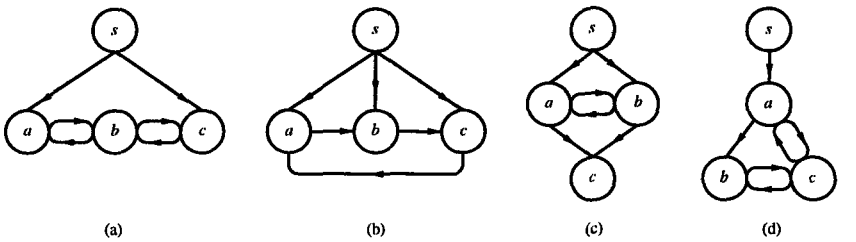


Fig. 5. Examples of extensions of the basic irreducible control flow graph of Fig. 2

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 Sect. 3.1 explains the node splitting technique. 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 8. 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, $Labels(P) = (label\,(x_1), \ldots, label\,(x_k))$. 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 Fig. 6 are equivalent. Node splitting is defined as:

Definition 9. *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. If node n has $K \geq 2$ predecessors and $L \geq 0$ successors then N' is defined as $N' = N \cup \{n_i \mid 2 \leq i \leq K\} - \{n\}$ and $E' = E - \{(p_i, n), (n, r_j) \mid 2 \leq i \leq K \wedge 0 \leq j \leq L\} \cup \{(p_i, n_i)(n_i, r_j) \mid 2 \leq i \leq K \wedge 0 \leq j \leq L\}$, 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 Fig. 6; node a of graph G_1 is split. Note that if a node n is split in the limit graph, then it is the corresponding node n in the original graph that must be split to remove irreducibility.

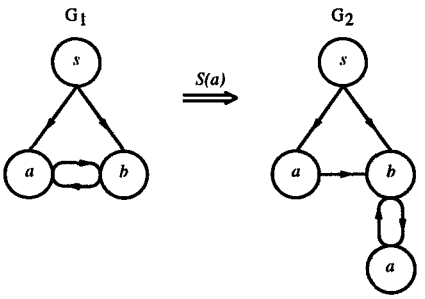


Fig. 6. A simple example of applying node splitting to node a

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

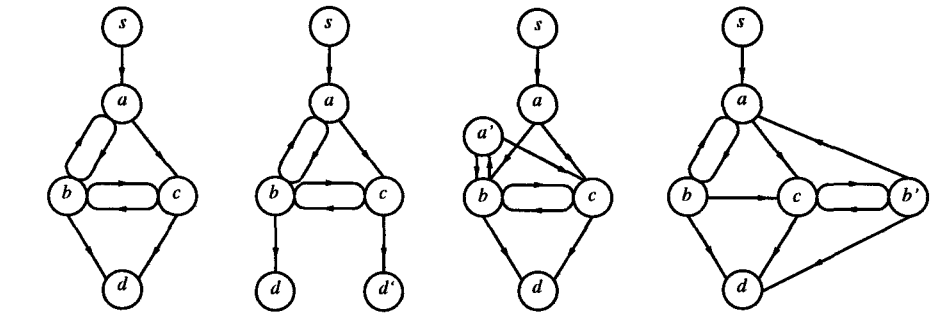
The transformation technique node splitting can be used to convert an irreducible control flow graph into a reducible control flow graph. From Hecht [7] we adopt Theorem 10.

Theorem 10. *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 Fig. 7a we see that the nodes a, b, c and d are candidate nodes for splitting. In Fig. 7b 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 Fig. 7c. Only splitting of node b or c converts the graph into a reducible control flow graph, see Fig. 7d.

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



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

Fig. 7. Examples of node splitting

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 11. 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 Fig. 7a $\{a, b\}$, $\{b, c\}$ and $\{a, b, c\}$ are loop-sets.

Definition 12. An *immediate dominator* of a node u, ID(u), is the last node on any path from the initial node s of a graph to u, excluding node u itself, which dominates u.

In Fig. 7a node s dominates the nodes s, a, b, c, and d, but it immediate dominates only node a.

Definition 13. A *Shared External Dominator set* (SED-set) is a subset of a loop-set L with the properties that it contains 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:

$$\text{SED-set}(L) = \{n_i \in L | ID(n_i) = d, d \notin L\} \tag{1}$$

Definition 14. A *Maximal Shared External Dominator set* (MSED-set) K is defined as:

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

The definition says that an MSED-set cannot be a proper subset of another SED-set. In Fig. 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 15. 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 Fig. 8a 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 the nodes of loop (b, c) do not form a SED-set.

In Sect. 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.

4.1 Controlled Node Splitting

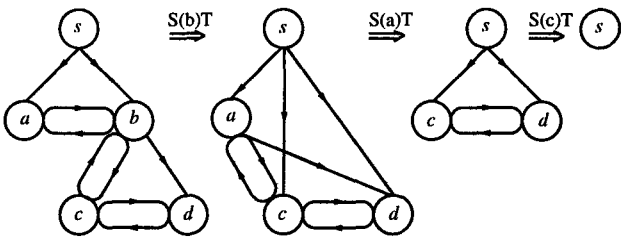
All nodes of an irreducible limit graph, except the initial node s of the graph, are possible candidates for node splitting since they have at least two predecessors. However splitting of some nodes is not efficient; see Sect. 3.2. CNS minimizes the number of splits. To accomplish this, two restrictions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

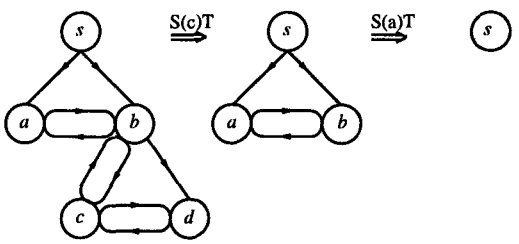
The first restriction prevents the splitting of nodes that are not in an SED-set. Splitting such a node is inefficient and unnecessary, as was demonstrated by Fig. 7b. These nodes are automatically reduced when all SED-sets are reduced to single nodes.

The second restriction is more complicated. The impact of this restriction is illustrated in Fig. 8. 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 Fig. 8a three splits are needed and in Fig. 8b only two. In Fig. 8a 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 Fig. 8a. Splitting an RC node merges the nodes of the MSED-set to which node RC belongs and the nodes of the MSED-set that

is dominated by the RC node into a single MSED-set. In [13] it is proven that splitting an RC node, and thus merging of MSED-sets, always leads to more splits.



a. Node splitting sequence of three nodes.



b. Node splitting sequence of two nodes.

Fig. 8. Graph with two different split graphs

Node splitting with above 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.

Theorem 16. *A MSED-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].

Theorem 17. *The minimum number of splits needed to reduce an MSED-set with k nodes is given by:*

$$T_{splits} = k - 1. \tag{2}$$

Theorem 18. *The minimum number of splits needed to convert an irreducible graph, with n MSED-sets, into a reducible graph is given by:*

$$T_{splits} = \sum_{i=1}^{n} (k_i - 1) \tag{3}$$

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

The proofs of both theorems are given in [13]; it is also shown that this minimum can be reached by obeying above restrictions. In Fig. 8 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 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 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 Fig. 9.

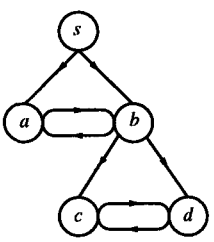


Fig. 9. A graph that has a common node that is not in the set RC

   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 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. A flow graph and its tree with all possible split sequences is drawn in Fig. 10. The nodes of the 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 each node contains 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, 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.

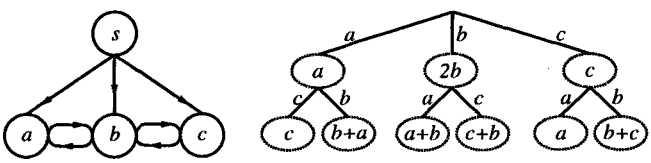


Fig. 10. An irreducible graph with its copy tree

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 transformations and splits, more formally $G^* = T(G)(S(G)T(G))^*$.

Theorem 19. *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:*

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

The proof of this theorem is given in [13].

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:

$$H(n_i) = Q(n_i) * (\text{\# predecessor nodes} - 1) \tag{4}$$

for every candidate node. This heuristic picks the node which results in the fewest copies. One can easily see that fewer predecessors nodes lead to fewer instructions to copy. The results of this heuristic, compared to the best possible split sequence, are given in Sect. 5.

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. 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 Sect. 4.1. This results in the minimum number of splits, but not in the minimum number of copied instructions.
- 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 number of copied instructions are listed in Table 1. The reported results of the methods UCNS, CNS and CNSH are the averages of all possible split sequences.

The first column in the Table 1 lists the procedure name, with the program name in parentheses. The second column gives the number of instructions of the procedure before an algorithm is applied. The other columns give the number of copied instructions that result from the algorithms. The absolute number of copies is given and a percentage that indicates the growth in code size with respect to the original number of instructions.

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 code size is 235.5%. CNS performs better, a growth of 30.1% 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 code size for both methods CNSH and ONS is 2.9%. 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 (re_search_2) with a very small difference. The execution time of CNSH is only slightly longer than for UCNS.

[2] We used a RISC like MOVE architecture. The MOVE project [14, 1] researches the generation of application specific processors (ASPs) by means of Transport Triggered Architectures (TTA).

Table 1. The number of copied instructions

Procedure(Progr.)	#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%)

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 235.5%. Controlled node splitting performs better (32.2%) 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 2.9%.

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. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 1994.
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.
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 Congress*, 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. Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. Technical Report JJ-9501, Delft University of Technology, Department of Electrical Engineering, 1995.
14. Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.