Handling Irreducible Loops: Optimized Node Splitting vs. DJ-Graphs

SEBASTIAN UNGER
and
FRANK MUELLER
North Carolina State University

This paper addresses the question of how to handle irreducible regions during optimization, which has become even more relevant for contemporary processors since recent VLIW-like architectures highly rely on instruction scheduling. The contributions of this paper are twofold.

First, a method of optimized node splitting to transform irreducible regions of control flow into reducible regions is formally defined and its correctness is shown. This method is superior to approaches previously published since it reduces the number of replicated nodes by comparison.

Second, three methods that handle regions of irreducible control flow are evaluated with respect to their impact on compiler optimizations. First, traditional node splitting is evaluated. Second, optimized node splitting is implemented. Third, DJ Graphs are utilized to recognize nesting of irreducible (and reducible) loops and apply common loop optimizations extended for irreducible loops.

Experiments compare the performance of these approaches with unrecognized irreducible loops that cannot be subject to loop optimizations, which is typical for contemporary compilers. Measurements show improvements of 1-40% for these methods of handling irreducible loop over the unoptimized case.

Optimized node splitting may be chosen to retrofit existing compilers since it has the advantage that it only requires few changes to an optimizing compiler while limiting the code growth of compiled programs compared to traditional node splitting. Recognizing loops via DJ Graphs should be chosen for new compiler developments since it requires more changes to the optimizer but does not significantly change the code size of compiled programs while yielding comparable improvements.

Handling irreducible loops should even yield more benefits for exploiting instruction-level parallelism of modern architectures in the context of global instruction scheduling and optimization techniques that may introduce irreducible loops, such as enhanced modulo scheduling.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*; E.1 [**Data**]: Data Structures—*graphs*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Compilation, control flow graphs, loops, irreducible flowgraphs, node splitting, reducible flowgraphs, code optimization, instruction-level parallelism

A preliminary version of this work was published in Euro-Par 2001, Topic 04: Compilers for High Performance [Unger and Mueller 2001]. Authors' addresses: Frank Mueller, North Carolina State University, CS Dept., Box 8206, Raleigh, NC 27695-7534, mueller@cs.ncsu.edu, phone: +1 (919) 515-7889, fax: -7925.

1. INTRODUCTION

Compilers rely on the assumptions that loops are identified to perform a number of code optimizations. Most loop optimizations have only been formulated for natural loops with a single entry point, the header, i.e., the sink of the backedge(s) of such a loop. Multiple entry loops cause irreducible regions of control flow that are typically *not* recognized as loops by traditional algorithms. These regions may result from goto statements in the source code but they may also be introduced by optimizations that modify the control flow, including global software pipelining and code replication techniques. As a result, loop transformations and optimizations to exploit instruction-level parallelism cannot be applied to such regions so that opportunities for code improvements may be missed.

Modern architectures that support instruction-level parallelism often rely on global instruction scheduling and software pipelining by the compiler to fully exploit its capabilities. In particular, very long instruction word (VLIW) architectures are beginning to emerge in modern architectures, such as in the Phillips TriMedia and the IA-64, which require aggressive instruction scheduling to exploit their performance [Hoogerbrugge and Augusteijn 1999]. On one side, software pipelining demands knowledge about the structure of a program, which contemporary compilers generally do not support for irreducible regions of code. On the other side, aggressive global instruction scheduling may result in branch reordering and code replication, which itself may introduce irreducible regions. Enhanced modulo scheduling may also introduce irreducible loops [Warter et al. 1992]. Other code optimization techniques, such as trace scheduling and profile-guided code positioning for optimizing compilers [Colwell et al. 1988; Fischer and LeBlanc 1991; Pettis and Hansen 1990], may be combined with code replication [Mueller and Whalley 1992; Mueller and Whalley 1995] or applied during binary translation [Bala et al. 2000], which can result in irreducible loops.

Common algorithms to detect loops only cover natural loops [Aho et al. 1986]. As a result, irreducible loops are generally missed and will be ignored during loop optimizations, which results in performance penalties for each loop iteration. An alternative to ignoring loops is given by node splitting [Aho et al. 1986]. In this approach, interval graphs are reduced step-by-step. If the resulting graph is non-trivial (of more than one node), a node with multiple predecessors is replicated so that the reduction process can proceed. Upon reaching a single node, the reduction sequence is reversed but the replicated nodes remain in place. This process transforms an irreducible control flow into a reducible one with only natural loops. To our knowledge, optimizing compilers do not tend to take this approach due to the increase in code size, which can be exponential in theory. This paper contributes a new approach of optimized node splitting, which is formally defined, proved correct and shown to be superior to previously published methods and evaluates its implementation. Recent publications provide a third approach by detecting the nesting hierarchy of irreducible and natural loops. This allows the application of loop optimizations by extending them to irreducible loops with multiple entries, thereby decreasing the likelihood of missed opportunities for optimizations. The resulting code does not suffer from increases in code size, unlike the last approach. This paper compares these three approaches in terms of overhead of the imple-

mentation and benefit in performance. It also contributes methods to extend loop optimizations for the second approach when irreducible loops are recognized.

2. TRADITIONAL NODE SPLITTING

Node splitting is based on a certain kind of interval analysis known as T1/T2-Analysis, which was used to check for the presence of irreducible regions in a flow graph [Aho et al. 1986]. It iteratively performs two transformations on the flow graph reducing it to a simpler one. These transformations are:

T1 Remove any edge that connects a node to itself.

T2 If any node has exactly one predecessor, then replace this node and its predecessor with a single new node. All edges to the predecessor node are connected to the new node, and all edges leaving one of the original nodes will now originate from the new *abstract* node.

If these transformations are applied as long as possible the resulting graph is called the *limit graph*. As shown by M. S. Hecht [Hecht 1977] this limit graph is independent of the order of transformations and the nodes subject to transformations. If the final graph is trivial, i.e., it has only one node, the original flow graph was reducible. If the limit graph is non-trivial, all of its nodes either have none or *more* than one predecessor.

The idea of node splitting is to define an additional transformation T3, which is applied if neither T1 nor T2 are applicable anymore. T3 is defined as follows:

T3 Choose any node with at least two predecessors. Duplicate this node so that there is one copy for each of them. Each of the predecessors is now connected to one of the copies, and all of the outgoing edges of the original node are duplicated for each copy.

After the application of T3, it is possible to use T2 again on these duplicated nodes since they now have only one predecessor. If this process is repeated the resulting limit graph is always trivial. If the above process is reversed, leaving the duplicated nodes in place, the result is a *reducible* flow graph that is equivalent to the original one. The entire process is illustrated in Figure 1. First, the transformation T2

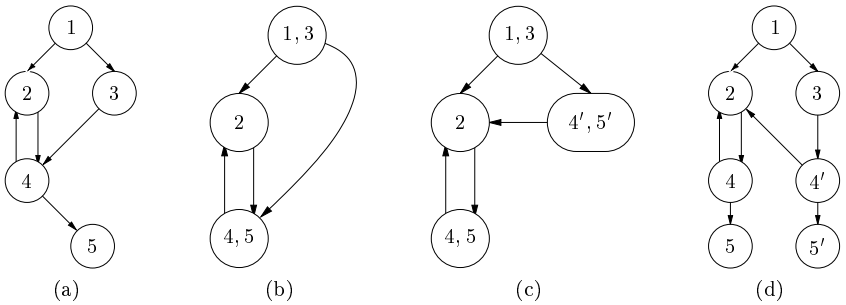


Fig. 1. Process of Node Splitting

is applied twice, once on node 3 and once on node 5. The result is shown in Figure 1(b). Since there are now no more nodes with only a single predecessor, T2 is no longer applicable. Therefore, T3 is applied to the abstract node containing nodes 4 and 5 with the result shown in Figure 1(c). Now the flow graph is reducible and the sequence T2((4′,5′)), T2((4,5)), T1((2,4,5)), T2((2,4,5)) reduces it into a single abstract node.

Reversing that process first leads to the same flow graph as in Figure 1(c). The transformation T3 that led to this graph, however, is not reversed but skipped. This means that the transformation T2(5) must now be reversed for two nodes 5 and 5′. Finally, the transformation T2(3) is reversed resulting in the final reducible flow graph of Figure 1(d).

Though this process already yields an algorithm, its performance may be poor. In Figure 1, for example, there is no need to split node 5. The resulting flow graph would still be reducible if there were just one node 5 with both 4 and 4′ as predecessors. This algorithm is inefficient because it does not consider which nodes form the irreducible loops. In this work, algorithms will be presented that exactly analyze the extent, structure and nesting of irreducible loops. Based on such an analysis a much better algorithm than that above will be constructed.

3. PROPERTIES OF IRREDUCIBLE REGIONS OF CODE

The motivation of this work is to develop an algorithm that converts an arbitrary irreducible control flow graph into an *equivalent reducible* one with the minimal possible growth in code size. This first involves the construction of an algorithm and second a proof of its correctness. This work builds on Sreedhar *et al.* [Sreedhar et al. 1996a], Bilardi and Pingali [Bilardi and Pingali 1996] as well as Janssen and Corporaal [Janssen and Corporaal 1997]. We reuse some of the terminology of Janssen and Corporaal [Janssen and Corporaal 1997]. In these previous suds, it was established that each irreducible loop has exactly one maximal subset of at least two of its nodes that have the same immediate dominator, which in turn is *not* part of the loop. They also discovered that these sets play an important role when minimizing the number of splits. Their definition of so-called Shared External Dominator sets was:

Definition 1 (Loop-set). *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.*

Definition 2 (SED-set). *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 (idom) is not part of* L*. A SED-set of a loop-set* L *is defined as:*

$$\textit{SED-set}(L) = \left\{ n_i \in L \;\middle|\; \text{idom}(n_i) = e \notin L \right\}.$$

For a definition of dominators, immediate dominators and other techniques for control flow analysis see, for instance, Muchnick [Muchnick 1997].

Definition 3 (MSED-set). *A Maximal Shared External Dominator set (MSED-*

set) K of a loop-set L is defined as:

$$\text{SED-set } K \text{ is maximal} \iff$$

$$\nexists \text{ SED-set } M, \text{ such that } K \subset M \text{ and } K, M \subseteq L.$$

For example, in the original flow graph of Figure 1, there is only one loop-set $\{2, 4\}$ and the MSED-set is the entire loop-set with 1 as external dominator.

The MSED-set is really a generalization of the single entry block in natural loops, and, in that special case, it consists of just one node. In the following, the nodes of MSED-sets will be simply called the header nodes or headers.

Building on that, new generalized definitions can also be found for the bodies (an irreducible loop can have more than one), backedges and the nesting of irreducible loops. Figure 2 illustrates this generalized structure. The domains represent the body of a natural loop. All edges from a domain back into the MSED-sets are backedges. The node e is the immediate dominator of the header nodes. The region called e-domain will be defined and used in the next section.

The following, generalized definitions of backedges and domains are based on MSED-sets whose definition in turn depends on the loop-set. This means, that the extension of the loop-set cannot be defined using backedges as it is for natural loops. This is only a problem because the definition of MSED-sets does in no way require the loop-set to be maximal. However, several of the following theorems only hold if the loop-sets are SED-maximal.

Definition 4 (SED-maximal loop-sets). *A loop-set L is SED-maximal if there is no other loop-set L' such that $L \subset L'$ and MSED-set$(L) \subseteq$ MSED-set(L').*

Based on that, domains and backedges can be precisely defined as:

Definition 5 (Domains). *Let L be an irreducible SED-maximal loop-set, K be its MSED-set and h_i be the nodes of K. The domain of h_i is then defined as:*

$$\text{domain}(h_i) = \{ n_j \in L \mid h_i \text{ dominates } n_j \}$$

Recall that the dominator relation is self-reflexive. Hence, a domain includes its header h_i.

Definition 6 (backedges). *Let L be an irreducible SED-maximal loop-set, K be its MSED-set and h_i be the nodes of K. An edge (m, n) with $m \in L$ and $n \in K$ is then called a back-edge of L.*

Figure 2 already suggested several properties of irreducible loops:

Theorem 1. *The nodes of L are in K or in exactly one of its domains.*

Theorem 2. *All edges into* domain$(h) \setminus \{h\}$ *originate from h.*

Theorem 3. *Let L_1 and L_2 be two different, SED-maximal loop-sets, K_1, K_2 their respective MSED-sets and e_1, e_2 the external dominators. Then*

—*If neither $L_1 \subset L_2$ nor $L_2 \subset L_1$ then $L_1 \cap L_2 = \emptyset$.* *(distinct loops)*

—*If $L_2 \subset L_1$ then there is a node $h \in K_1$ such that $L_2 \subset$* domain(h). *(nested loops)*

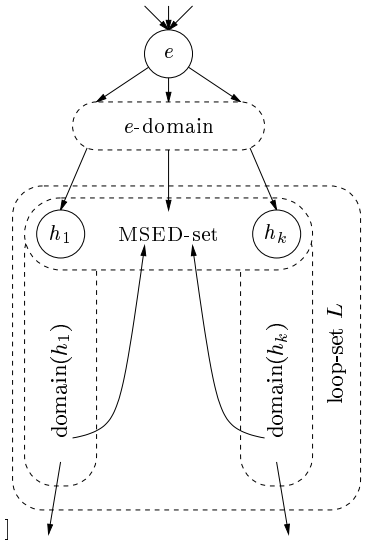


Fig. 2. Structure of Irreducible Loops

The proofs of these theorems can be found in [Unger 1998]. The results are used in the following section to develop an optimized algorithm for node splitting.

4. OPTIMIZED NODE SPLITTING

The knowledge about the structure of irreducible loops can be used to guide the T3 transformation to some extend. Repeated application of T1/T2 will collapse domains into their headers leaving an MSED-set. Applying T3 to a node in the MSED-set then splits a header and its *entire* domain (see Figure 3 where b is chosen). Alternatively, a could have been split leading to a minimal result if the

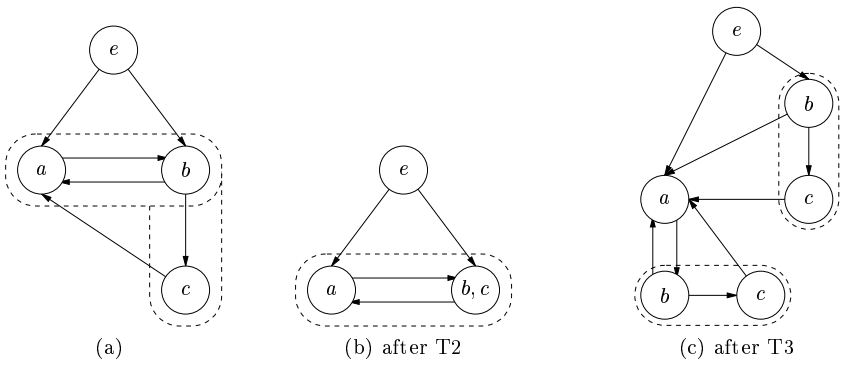


Fig. 3. Sample Split

weight $\sigma(a) < \sigma(b) + \sigma(c)$, which may be a function of code size. If, on the other

hand, b had three incoming edges (i.e., add a header node in the example) then three copies would result from splitting b and splitting a would have been preferable if $\sigma(a) < 2(\sigma(b) + \sigma(c))$. Hence, nodes should be split from the lightest to heaviest wrt. weights of abstract nodes (the headers and domains) *times* the number of incoming edges.

As the domains are collapsed into one abstract node, multiple edges from one domain to a single header node will reduce to just one edge from the abstract node to the header node. This is important because it reduces the number of copies of that node and is also true for multiple edges from the outside. Figure 2 suggests by the naming that the region called e-domain (defined below) should be handled just as any other domain. That is, transformations T1 and T2 should collapse it into e, thereby reducing multiple edges from that domain to any header node into one edge. Of course, T3 should not be applied to this abstract node.

Definition 7 (e-domain). *Let L be an irreducible SED-maximal loop-set, K be its MSED-set and e the external dominator. That is: If e is the immediate dominator of the nodes in K, then the set* e-domain *is defined as:*

$$e\text{-domain} = \left\{ n_i \in N \;\middle|\; \begin{array}{l} e \text{ dominates } n_i,\ n_i \notin L \text{ and} \\ \exists \text{ a path } p \text{ from } n_i \text{ into } L \text{ with} \\ e \notin p. \end{array} \right\}$$

Does this algorithm always produce the minimal reducible equivalent flow graph as in the example? Unfortunately not, as is shown in Figure 4. Node b_3 could

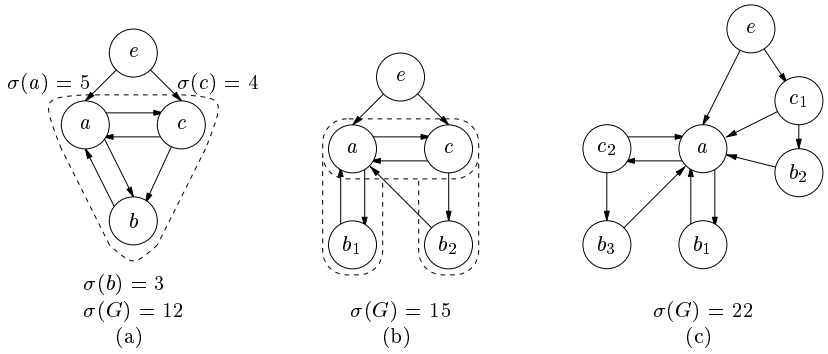


Fig. 4. Splitting Lightest Nodes First (Nodes with same labels have indices for better reference)

actually be avoided. Its predecessor node c_2 could as well jump to the copy b_1 as shown in Figure 5(c). Had not b but c been split first, Figure 5(c) would have been the result. This means that selecting the nodes to split just by their weight is not sufficient. In fact, greedy algorithms for selecting a node to split are not optimal since we can construct a counter-example in analogy to Fig. 4. Another question is if there is always an order that leads to the minimum. Alas, not even that is true. Figure 6 gives a flow graph together with its minimal reducible equivalent graph

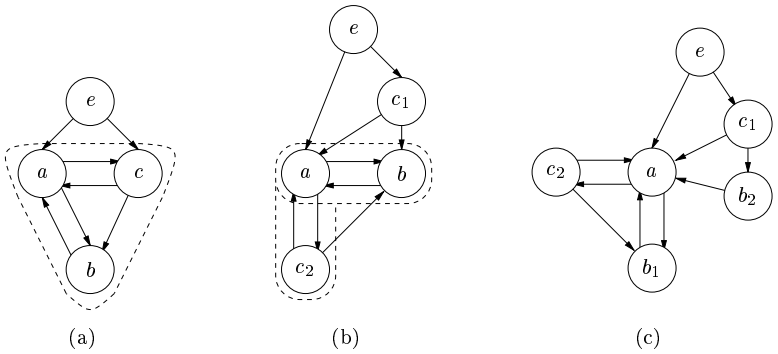


Fig. 5. Best Splitting Sequence (c, b)

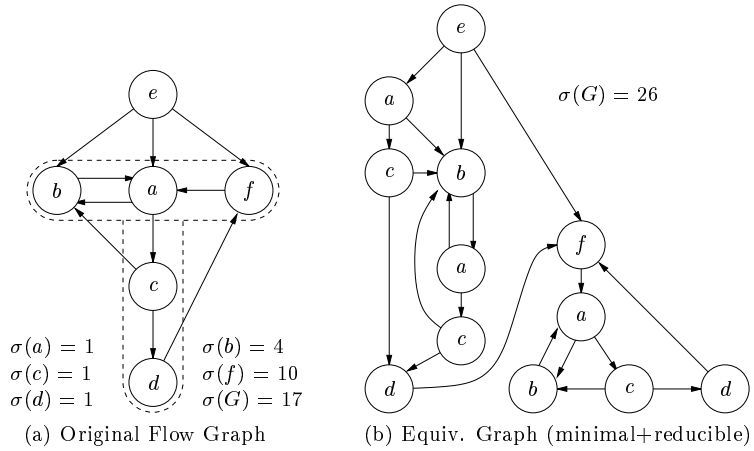


Fig. 6. T3 Cannot Split this Graph in a Minimal Way.

where no order will split the nodes to yield a minimal graph. The problematic node is node d. This node belonged to the domain of a. Therefore, by the algorithm above, each copy of a will get its own copy of d. But in the minimal flow graph two copies of a share the same node d. The problem is that d is only a part of the loop-set because of its edge to f. Once it has been moved out of the loop containing f, it no longer is within any loop and, therefore, it no longer is in the domain of a. This means that the domains may change in the process of splitting nodes and this cannot be handled by the simple algorithm above.

A new approach has been developed, based on the observation that all of the examples contained one of the header nodes that was not split at all. In the previous approach, the header nodes had been selected for splitting one after another until only one remained. At that moment, the irreducibility had been resolved. However, as we have seen, there were examples where no selection scheme led to a minimal

result. The new approach, therefore, does not choose single nodes for splitting. Instead, it chooses a single header node (plus its domain, of course) that should be the one that is *not split at all*. All other nodes of the loop-set are split once. This is illustrated in Figure 7(a). The regions containing the copies of the remaining nodes of L are not yet guaranteed to be reducible but they are guaranteed to be smaller than L by at least one node. Hence, the above step can be applied recursively to these copied regions. This new approach also needs a scheme for selecting the node h_1 with the advantage over the previous approach that for any flow graph there is a selection scheme that leads to a minimal result. All that remains is to actually find this scheme.

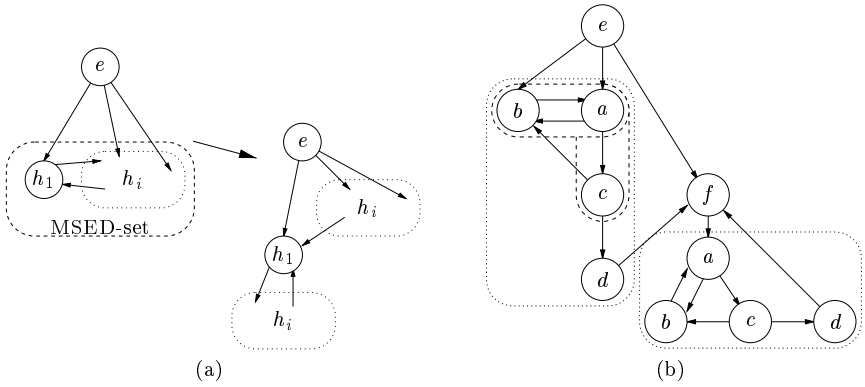


Fig. 7. One Step of the Recursive Algorithm

4.1 Outline of the Algorithm

In the following, the algorithm is defined more precisely. The following notation will be used in the following: If f is a function over the nodes of any control flow graph, then $f(X)$, where X is a subset of these nodes, stands for the set $\{f(x)|x \in X\}$.

Definition 8 (Transformation $\mathbf{T}_r$). *Let $G = (N, E, s)$ be an arbitrary (irreducible) control flow graph, L an SED-maximal, irreducible loop-set of G, K its MSED-set, e the external dominator and h an arbitrary node from K. Then the transformation $G' = (N', E', s') = T_r(G, L, h)$ is defined as follows (with $S = (L \setminus \text{domain}(h))$):*

—$N' = (N \times \{1\}) \cup (S \times \{2\})$

—$E' \subset N' \times N'$ *such that the following restrictions hold:*

—$(x, y) \in E \wedge (x, y) \notin (\text{domain}(h) \times S) \iff ((x, 1), (y, 1)) \in E'$ *(8.1)*

—$(x, y) \in E \wedge (x, y) \in (\text{domain}(h) \times S) \iff ((x, 1), (y, 2)) \in E'$ *(8.2)*

—$(x, y) \in E \wedge (x, y) \in (S \times (N \setminus S)) \iff ((x, 2), (y, 1)) \in E'$ *(8.3)*

—$(x, y) \in E \wedge (x, y) \in (S \times S) \iff ((x, 2), (y, 2)) \in E'$ *(8.4)*

—$s' = (s, 1)$

The above transformation represents one step of the algorithm. All nodes of the loop-set L, except for those in the selected header node's domain, are split. The new copies of these nodes are represented by the syntactical construction $S \times \{2\}$ while the old nodes are represented by $N \times \{1\}$. In other words, a single T_r transformation results in one split to copy S so that unnecessary copies are avoided.

For example, applying T_r once on the graph of Fig. 6(a) without copying node f results in Fig. 7(b). Then, $domain(f) = \{f\}$ and $S = \{a, b, c, d\}$. Restriction 8.1 preserves the original edges, such as $e \to f$. Restriction 8.2 creates new edges from the domain to loop nodes outside the domain, such as $f \to a$. Restriction 8.3 duplicates exits from the loop to original nodes (not depicted in the example). And restriction 8.4 covers internal nodes within the duplicated region, such as $a \to b$ in the lower right box of Fig. 7(b).

A proof of the correctness of optimized node splitting using T_r is given in the following. The formal proof shows the correctness but it does not provide a constructive scheme to select the optimal header node excluded from splitting. Intuitively, we can yield optimal results by freezing the "right" domain in each step, thereby minimizing the number of splits, for a clairvoyant selection of the header of this domain. It can be shown that an optimal header exists and that T_r would result in an optimal splitting sequence if the optimal node was chosen. However, it is an open problem if an efficient (polynomial time) algorithm to identify this node exists. In our experiments, we use the heuristic to select (and then freeze) the *heaviest* header and its domain while all other nodes in the loop set are split. The heaviest header is the node of the MSED-set whose entire domain has the largest number of instructions within the intermediate code representation (see section 6).

The pseudo code of the recursive algorithm for T_r is given in the appendix since it uses data structures defined by DJ graphs, which are introduced in more detail in section 5.

4.2 Proof of Correctness

The labeling and weight of the new graph can be easily defined by the labeling and weight of the original graph:

Definition 9 (Corresp. Labeling and Weight). *If l is an labeling of G, then the* corresponding labeling *l' of G' is defined as*

$$l'((x, \cdot)) = l(x)$$

If σ is a weight function of G, then the corresponding weight function *σ' of G' is defined as*

$$\sigma'((x, \cdot)) = \sigma(x)$$

Here, the notation $(x, \cdot)$ stands for any tuple (x, y).

For the proof of correctness we assume no particular order in which T_r is applied. The algorithm just chooses an SED-maximal, irreducible loop-set L and a header node h of L and applies $T_r(G, L, h)$. Then, the algorithm will finally construct an equivalent reducible flow graph, where $\sim$ denotes the equivalence relation between two graphs, *i.e.*, their labeling correspondence for all nodes. Because $\sim$ is transitive, it suffices to show that the following theorem holds in order to show the partial correctness of the algorithm.

Theorem 4 (Partial correctness). *Let G be an arbitrary (irreducible) control flow graph, L an SED-maximal, irreducible loop-set of G, K its MSED-set, e the external dominator and h an arbitrary node from K. Then $G \sim T_r(G, L, h)$ with respect to the labeling l and the corresponding labeling of $T_r(G, L, h)$.*

PROOF. $\Longrightarrow$
Let $p = (p_1, \ldots, p_n)$ be an arbitrary path of G. Then $q = (q_1, \ldots, q_n)$ with

$$q_1 = (p_1, 1)$$
$$q_{i+1} = \begin{cases} (p_{i+1}, 2) & \text{if } p_{i+1} \in S \wedge (q_i = (p_i, 2) \vee \\ & \qquad p_i \in \text{domain}(h)) \\ (p_{i+1}, 1) & \text{otherwise} \end{cases}$$

is a path of $\mathrm{T}_r(G, L, h)$ with $\forall 1 \leq i \leq n : l(p_i) = l_r(q_i)$ where l_r is the corresponding labeling of $\mathrm{T}_r(G, L, h)$.

$\Longleftarrow$
Let $q = (q_1, \ldots, q_n)$ be a path of $\mathrm{T}_r(G, L, h)$. From the construction of $\mathrm{T}_r(G, L, h)$ follows that $q_i = (p_i, k)$ with $k \in \{1, 2\}$ and if $(q_i, q_{i+1}) \in E'$, then $(p_i, p_{i+1}) \in E$. Thus, $p = (p_1, \ldots, p_n)$ is a valid path of G and by Definition 9 $l_r(q_i) = l(p_i)$, where l_r is again the corresponding labeling. □

If the algorithm applies T_r as long as there is any irreducible loop-set, then the proof of reducibility and termination become equivalent. Therefore, to complete the proof of correctness, it is sufficient to show that T_r finally constructs a reducible flow graph.

Theorem 5 (Termination). *Let $G = (N, E, s)$ be an arbitrary (irreducible) control flow graph. Then, if G is repeatedly transformed by $T_r(G, L, h)$, where L is an arbitrary irreducible loop-set of G and h is a header node of L, then after a finite number of such transformations G will be reducible.*

PROOF. From Theorem 1 follows that for each irreducible loop-set L' with $L' \neq L$ one of the following is true:

—$L' \cap L = \emptyset$
But then the nodes and edges of L' are completely unaffected by the application of T_r.

—$L' \supset L$
But then $L \subset \text{domain}(h')$ for some $h' \in \text{MSED-set}(L')$ and the changes made by T_r are restricted to domain(h') and do not affect MSED-set(L').

—$L' \subset L$
Then $L' \subset \text{domain}(h')$ for some $h' \in \text{MSED-set}(L)$. If $h' = h$, then L' is completely unaffected by the application of T_r. Otherwise $L' \subset S$ and thus has been split. However, since all edges *within* S have been duplicated on both copies $S \times \{1\}$ and $S \times \{2\}$, L' is present twice in $\mathrm{T}_r(G, L, h)$ and both copies still have the same number of header-nodes.

However, since $(h, 1)$ dominates all nodes in $S \times \{2\} \cup \text{domain}(h) \times \{1\}$ and there is no edge from $S \times \{2\} \cup \text{domain}(h) \times \{1\}$ to $S \times \{1\}$, any remaining irreducible loop-set in these regions is either a nested loop-set already present in G or (since

all of the original header nodes still dominate their respective domains) this loop set must have at least one less header node than L.

Thus, on every application of T_r one loop-set's MSED-set becomes smaller by one node, while the MSED-sets of all other loop-sets are unaffected. Though some loop-sets are duplicated, these can only be subsets of the one that got smaller. And finally, each irreducible loop-set will become a top-level one and then can itself only become smaller. Therefore, all loop-sets will eventually have an MSED-set with only one node and thus will be reducible. □

This means that any flow graph can be converted to an equivalent and reducible one by the repeated application of T_r alone. It remains to be shown that there is always a sequence of transformations that leads to a minimal final graph. To prove this, a sequence is constructed from an arbitrary minimal graph. This, of course, is not a constructive proof as it cannot be used to transform a flow graph into a minimal equivalent one without knowing the final graph beforehand. However, it is still important to prove the above property.

In the following, a brief outline of the remaining steps of the proof is given. The complete formal proof is presented in the appendix.

The proof consists of the following steps: A minimal sequence is constructed by choosing a loop L of flow graph G within some constraints. This L is identified within the minimal, reducible equivalence graph G_{min} to find the header h for the transformation $T_r(G, L, h)$, which is accomplished by describing equivalent loop sets of G and G_{min} via their labeling. Then it is shown that

—there is always a loop-set L with exactly one equivalent loop-set;

—a header of the graph after applying T_r is always a header node of L; and

—the final (reducible) graph G from this construction is minimal wrt. its weights since $\sigma(G) = \sigma_{min}(G_{min})$.

5. USING DJ GRAPHS TO OPTIMIZE IRREDUCIBLE LOOPS

The representation of DJ Graphs [Sreedhar et al. 1996a] may be used for incremental data-flow analysis but it also provides the means to perform loop optimizations on irreducible loops. By constructing the DJ Graph of a control-flow graph, natural and irreducible loops and their nesting hierarchy can be detected.

An example is depicted in Figure 8, which represents the DJ-Graph of the control-flow graph in Figure 6(a). The DJ-Graph consists of the edges of the dominator tree (dashed), backedges, and the remaining edges of the control flow called cross edges (solid). Furthermore, sp-back edges are control-flow edges $x \rightarrow y$ where $x = y$ or y is an ancestor of x in a spanning tree resulting from a depth-first search. In the example, a search in the order of the indicies of the nodes indicates that the edges marked with bullets are sp-back. Loops in the DJ-Graph can then be found starting from the lowest dominator level (level 3). If a backedge exists at the current level, then nodes corresponding to its natural loop are collapsed into one node. Afterwards, if a cross edge is also sp-back, all strongly connected components at the current level or below represent an irreducible loop and are collapsed to a single node before considering the next higher level. In the example, there are no backedges but several cross edges at level 1 that are also sp-back. The only

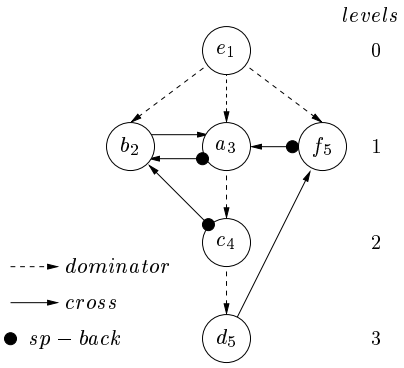


Fig. 8. DJ-Graph for Fig. 6(a)

strongly connected components comprise all nodes at level 1 or below, *i.e.*, exactly one irreducible loop is found. However, Figure 6(b) shows that an inner loop $\{a, b\}$ and an outer loop $\{a, b, c\}$ may be distinguished by optimized node splitting. Nonetheless, DJ-Graphs still allow the distinction of irreducible loop bodies either if they comprise different levels or if they represent distinct strongly connected components. Furthermore, DJ-Graphs also allow the detection of reducible loops within irreducible ones. Had there been an edge $d \rightarrow c$ in Figure 8, then this edge would have been recognized as a backedge whose source and sink comprise a loop at level 2.

There are other differences between natural loops and DJ-graphs representations of irreducible loops. Instead of one loop header for natural loops, irreducible loops have multiple entry blocks with predecessor blocks outside the loop. Furthermore, there is no block in an irreducible loop that dominates all other blocks within the loop. Notice, however, that we allow a natural loop to share a header with an irreducible loop. We still distinguish both loops in this case. These differences require changes to other loop optimizations, as presented in the following.

5.1 Adapting Code Motion

Code motion moves invariant operations out of the body of a natural loop into the preheader block. For irreducible loops, the set of entry blocks can be augmented by a set of preheader blocks. Then, a copy of a loop-invariant operation is moved into all preheaders at once. Code motion as stated in [Aho et al. 1986] applies with minor changes, *e.g.*, to find invariant statements:

(1) $dst = src$ is invariant if src is constant or its reaching definitions are outside the loop. We check registers live on entry for *each* preheader of the loop in this case.

(2) transitively mark statements in step 3 until no more unmarked invariant statements are found.

(3) $dst = src$ is invariant if src is constant, if its sole reaching definition inside the loop is marked invariant or if its reaching definitions are outside the loop.

The optimizing compiler VPO [Benitez and Davidson 1988] used in the experiments also moves invariant memory accesses but reads are only moved if there are no writes within the loop and the reads are executed during each iteration. The latter condition prevents that reads conditional on non-null references are moved out of the loop without the corresponding test-for-null, which would have caused a memory fault. For natural loops, we test for an unconditional execution during each loop iteration by requiring a read to dominate all sources of backedges within the loop. Since irreducible loops do not have backedges, we have to calculate equivalent information beyond DJ Graphs. For each entry of an irreducible loop, we delete all other entries and collect the sources of all backedges within the resulting region. Notice that such a region may contain more than one natural loop now. We call the collected blocks the sources of *pseudo-backedges* of the irreducible loop. A block of an irreducible loop is executed during each iteration if it dominates all sources of pseudo-backedges within the corresponding reducible regions. This requires that the dominator information of the reducible pseudo-regions be associated with an irreducible loop. In Section 7, other techniques for code motion are discussed with regard to their applicability to irreducible code.

5.2 Handling Induction Variables

Finding induction variables becomes more complicated due to irreducible loops. We limit our approach by requiring that changes to induction variables are performed in blocks which are executed on each loop iteration. This information is already available from code motion for memory reads.

In addition, one could allow balancing modifications in corresponding conditional arms. These arms range from a split at an always iterated block to a join at the next block that is always executed during each loop iteration. We did not implement this extension.

Once induction variables are identified, strength reduction and induction variable elimination can be performed as for natural loops, except that invariant operations of register loads are moved into *all* preheaders of the irreducible loop.

5.3 Other Optimizations

Similar to the handling of induction variables, recurrences can be optimized by moving the prologue into all preheaders, given that the memory access originates in a block that is executed on each loop iteration. Other optimizations also benefit from the additional loop information. For example, global register allocation is performed by prioritized graph coloring in VPO. The priority is based on the loop frequency, which is readily available even for irreducible loops and their nesting within other loops. No modification was required to such optimizations.

6. MEASUREMENTS

We chose VPO [Benitez and Davidson 1988] as a platform to conduct a performance evaluation. VPO only recognizes natural loops with a single header where a loop is entered, just as all contemporary optimizing compilers we know of. Irreducible regions of code (including natural loops within this region) are recognized but, due to lack of knowledge about their structure (headers, backedges and exits), loop optimizations cannot be applied.

First, we added the recognition of DJ Graphs to VPO, extended code motion, strength reduction, induction variable elimination and recurrences. Second, we also implemented the method of optimized node splitting through the T_r transformation described in this paper. The heuristic driving node selection was to choose the header of the domain with the most instructions. This node (and its domain) were not split while all other nodes in the loop set were split. Third, traditional node splitting using the T1/T2/T3 transformations was also integrated. The heuristic considers for each header the number of instructions times predecessors (see section 4, §1). The header with the smallest heuristic value is then chosen. These optimizations were activated through additional compiler switches.

A number of test programs with irreducible loops were used to measure the effect of the three different approaches. Dfa is a program that simulates a deterministic finite automaton representing an irreducible loop containing two independent natural loops (see Fig. 2a in [Sreedhar et al. 1996a]). Arraymerge merges two sorted arrays into a third sorted one and was originally extracted and translated from a Fortran application. Tail (output the bottom of a file), Unifdef (C-files without conditionally compiled regions), Hyphen (find hyphenated words), Cpp (preprocessor for C-files), Nroff (text formatter) and Sed (string editor) are UNIX utilities.

The measurements were collected for the Sun SPARC architecture using the environment for architectural study and experimentation (EASE) [Davidson and Whalley 1990] that is integrated into VPO. Table I shows in column 1 the name of a program with an irreducible region in some function. (See Table III for a correlation between functions and programs.) Column 2 depicts the number of dynamically executed instructions (of the object code) within this function when loop optimizations are not applied to irreducible regions. Column

	Ignore	DJ	Node Splitting	
Program	IRLoops	Graph	T3(trad.)	T_r(opt.)
dfa	64736	-13.91%	-13.85%	-13.88%
arraymerge	37401300	-36.76%	-39.70%	-39.70%
tail	270012	0.00%	0.00%	0.00%
unifdef1	16864	-0.36%	+9.01%	+10.37%
unifdef2	13169	-4.40%	-7.23%	-1.10%
hyphen	447232850	+0.01%	+0.01%	+0.01%
cpp	7510602	+0.05%	-1.61%	-1.18%
nroff1	340830	-8.28%	-7.19%	-13.50%
nroff2	1950909	-4.37%	-4.42%	-0.19%
nroff3	350305	0.00%	+0.06%	+0.13%
sed	846	-0.59%	-4.02%	-4.49%

Table I. Executed Instructions and their Changes for Irreducible Regions

3 represents the portion of executed instructions for the DJ-Graph approach and adapted loop optimizations relative to Column 2. Column 4 shows the portion of executed instructions for traditional node splitting (using the T3 transformation) relative to Column 2. Column 5 depicts the portion of executed instructions for optimized node splitting (using the T_r transformation) relative to Column 2.

For the DJ-Graph approach, optimized and traditional node splitting show comparable reductions of 1-40% in the number of executed instructions. Notice that

our compiler, VPO, optimizes code quite aggressively, *i.e.*, in the presence of irreducible regions only *loop* optimizations are suppressed. All other optimizations are still performed on reducible and irreducible regions. Furthermore, loop optimizations *are* performed on reducible loops that share a function with irreducible ones. Other compilers may not optimize *any* loops within a function when an irreducible region is detected, or may even suppress a larger range of optimizations in such a case. In that case, handling irreducible code would result in even higher gains. The quantity of improvements are subject to the execution frequency of irreducible regions within the enclosing function. For example, Arraymerge contains a central loop for sorting that was irreducible. Tail, on the other hand, contains an irreducible loop for block reads, which is executed infrequently relative to the other instructions within the function. The function "skipcomment" in Unifdef1 showed worse results for optimized node splitting, which is correlated to fewer delay slots of branches being filled. The measurements only include those instructions in delay slots that are actually executed [Weaver and Germond 1994]. But when no instruction can be moved into a delay slot due to instruction dependencies, then no-ops must be placed in the delay slot resulting in the execution of additional instructions. A more aggressive method to fill delay slots may be required in this particular case to compensate for the increased number of branch instructions due to node splitting. Similar effects were observed for cases where little changes were observed.

Table II depicts for a function of a program containing an irreducible loop the size of the function in number of instructions. The code size only changes insignificantly

	Ignore	DJ	Node Splitting	
Program	IRLoops	Graph	T3(trad.)	T_r(opt.)
dfa	167	-4.79%	+30.54%	+21.56%
arraymerge	120	-0.83%	+32.50%	+19.17%
tail	470	+1.70%	+6.60%	+3.83%
unifdef1	56	+7.14%	+26.79%	+28.57%
unifdef2	56	+1.79%	+21.43%	+25.00%
hyphen	168	+8.93%	+20.24%	+20.24%
cpp	730	-0.27%	+1.64%	+2.33%
nroff1	239	+0.84%	+35.15%	+10.46%
nroff2	215	+0.47%	+25.58%	+1.86%
nroff3	102	0.00%	+16.67%	+9.80%
sed	1260	+2.06%	-0.40%	-1.27%

Table II. Size of Irreducible Regions in Instructions and their Changes

for DJ-Graphs. These small changes are due to other optimizations. The quantity of changes depends on the number of preheaders and the compensation by other optimizations, such as peephole optimization. For optimized node splitting, the code size changes between -1% and 28%. This change in size is measured relative to the original function containing an irreducible loop. The change in code size relative to the *entire program* was between 0.5% and 3.5% for larger test programs and 8-17% where the irreducible loop comprised most of the test program (Dfa, Arraymerge and Hyphen). The fact that node splitting stops at function boundaries limits the overall increase in code size for the entire program so that exponential growth was

not encountered in the experiments and is unlikely in general. Traditional node splitting resulted in more code growth (up to 35%). In most cases, traditional node splitting not only resulted in larger code, the dynamic instruction count was also not much different than optimized node splitting. This shows that for most cases optimized node splitting reduces the amount of code duplication while preserving the performance. In a few cases (e.g. unifdef2), the traditional method performed better than the optimized one (few instructions executed and less duplication). This seems to indicate that there is still room for further investigation into better heuristics or even profile-driven node selection for the T_r transformation.

The differences between the two node splitting techniques are further illustrated in Table III depicting the number of copied register transfer lists (RTLs) [Benitez 1991] for traditional T3 splitting (Column 2) and optimized T_r splitting (Column 3) with changes relative to Column 2 in parenthesis. Since both node

	Node Splitting	
Program(Function)	T3(trad.)	T_r(opt.)
dfa(main)	701	204 (-70.90%)
arraymerge(MergeArrays)	306	50 (-83.66%)
tail(main)	1906	100 (-94.75%)
unifdef1(skipcomment)	218	56 (-74.31%)
unifdef2(skipquote)	191	44 (-76.96%)
hyphen(main)	914	153 (-83.26%)
cpp(cotoken)	2791	71 (-97.46%)
nroff1(text)	975	138 (-85.85%)
nroff2(getword)	787	103 (-86.91%)
nroff3(suffix)	417	28 (-93.29%)
sed(fcomp)	4897	38 (-99.22%)

Table III. Number of Copied RTLs (Instructions) during Node Splitting

splitting approaches are performed as one of the first optimizations, each RTL of the intermediate code representation resembles a very simplistic instruction. The numbers show that the traditional method results in significantly more replicated code after node splitting than the optimized approach, which only requires 1-30% of copied RTLs under T_r relative to T3. This underlines the qualities of our new method for node splitting. Notice that the increases in code size depicted in Table III are not as significant as the differences between T3 and T_r in Table II. This can be explained as follows. Both node splitting techniques are performed as one of the first code optimizations. VPO then thoroughly optimizes the resulting code, *e.g.*, through aggressive peephole optimization, folding branch chains etc., which results in a less significant difference of increased code sizes for T3 and T_r. This suggests that T3 may yield considerably inferior results than T_r for optimizing compilers with less aggressive optimizations than VPO. It shows again that optimized node splitting is superior to the traditional approach but actual savings depend on the phase ordering of optimizations and the infrastructure of the optimizing compiler as such.

In addition, we compared the node splitting methods T3 and T_r with the controlled node splitting (CNS) using heuristics by Janssen and Corporaal [Janssen and Corporaal 1997]. The CNS approach is detailed in the related work

section. The measurements indicated that CNS differed only insignificantly from our T3 approach, both in the number of executed instructions and the change in code size. Careful analysis revealed that the heuristic used for T3 almost always picked the same nodes for splitting as CNS. Further restrictions on node selection by CNS only occurred in one case (nroff) but had hardly any effect on the results.

We also measured the instruction cache performance for a 4kB and 512B direct-mapped cache using VPO and EASE. The hit ratio did not change significantly (less than 1%) for the tested programs, regardless of the cache size. For changing code sizes, the cache work is often a more appropriate measurement [Mueller 1991], where a miss accounts for 10 cycles delay (for going to the next memory level [Hennessy and Patterson 1996]) and a hit for one cycle. The methods of handling irreducible loops all resulted in reduced cache work for most cases, varying between a reduction of 6% and 28%. This reduction seems to indicate that execution in replicated regions tends to be localized, *i.e.*, once such a region is entered, executing progresses within this replica rather than transferring control between different replicas.

	Ignore	DJ	Node Splitting	
Program	IRLoops	Graph	T3(trad.)	T_r(opt.)
dfa	162007	-5.56%	-5.53%	-5.55%
arraymerge	53911598	-19.38%	-22.54%	-27.54%
unifdef	251607	-0.25%	+0.34%	+0.64%
cpp	18525543	+0.02%	-0.65%	-0.48%
nroff	44628332	-0.31%	-0.41%	-0.27%

Table IV. Executed Instructions and their Changes for Entire Program

Finally, the impact of handling irreducible loop on *entire* programs was assessed. Table IV depicts the change in executed instructions results for the different optimization methods. The results illustrate that the benefits within a function (as seen in Table I only translate to the entire program if irreducible loops are frequently executed, such as for Dfa and Arraymerge, while the overall impact on other test programs was low. The changes for Tail, Hyphen and Sed were omitted since they were below a tenth of a percentile for all three approaches. Notice that handling irreducible loops does not inadvertently impact the compilation overhead. In fact, detecting the presence of irreducible loops can be performed without additional overhead during the recognition of reducible loops [Aho et al. 1986]. Hence, compilation overhead is only imposed if an irreducible loop is actually present. Table V depicts the change in program size of the entire program for the optimization methods in question. As expected, changes in program size are less significant than the reported changes in size for functions (see Table II), which indicates that code size becomes less significant when considering the overall program.

7. RELATED WORK

Reducible flow graphs were first mentioned by Allen [Allen 1970]. The idea of node splitting stems from Cocke and Miller [Cocke and Miller 1969]. DJ Graphs are due to Sreedhar *et al.* [Sreedhar et al. 1996a]. Loop analysis via DJ Graphs

	Ignore	DJ	Node Splitting	
Program	IRLoops	Graph	T3(trad.)	T_r(opt.)
dfa	208	-3.85%	+24.52%	+17.31%
arraymerge	283	-0.71%	+28.27%	+8.13%
tail	507	+1.58%	+4.73%	+3.55%
unifdef	943	+0.53%	+4.77%	+3.18%
hyphen	219	+6.85%	+15.53%	+15.53%
cpp	4606	-0.13%	+0.26%	+0.37%
nroff	12709	-0.13%	+1.12%	+0.42%
sed	4360	+0.57%	-0.11%	-0.37%

Table V. Size of Entire Programs in Instructions and Changes

has the advantage over previously proposed methods that it finds nestings of natural and irreducible loops in arbitrary orders. DJ Graphs have also been used to perform incremental data-flow analysis for irreducible control flow. A framework for graph reduction and variable elimination is described by Sreedhar *et al.* [Sreedhar et al. 1996b].

Even more recently, Havlak proposed a method for recognizing reducible and irreducible loops as well as the nesting of either ones [Havlak 1997]. His algorithm used node splitting only for headers of natural loops contained within irreducible loops as a means to have distinct header nodes. This work did not use node splitting to make irreducible loops reducible, whereas our work did. Furthermore, our notion of backedges is independent of any graph traversals while Havlak's backedges for irreducible loops depend on the order of a traversal of the control flow. Ramalingam [Ramalingam 1999; Ramalingam 2000] contributed performance improvements and a common formal framework for three schemes for recognizing loop structures, including those by Sreedhar *et al.* and Havlak. Since our study is concerned with the performance of the *compiled programs* rather than the performance of the *compiler*, we did not implement his improvements. However, we strengthen the results of [Ramalingam 2000] through our structural definition of loops and our SED-maximal loop sets that capture the loop descriptions of previous work and represent a *minimal loop nesting forest*. In particular, we reduce irreducible loops into reducible ones in a bottom-up fashion (wrt. the level in the dominator tree) by isolating (and freezing) the largest domain and its header while splitting the remaining nodes in the loop set. Recursive splitting ensures that different loops within one irreducible region can be isolated. Hence, we go beyond the approach by Sreedhar *et al.* although our algorithm uses the same data structures. We also showed how several optimization methods for reducible graphs can be transformed into methods for irreducible graphs, which, once again, strengthens Ramalingam's results [Ramalingam 2000].

The notion of MSED-sets is introduced by Janssen and Corporaal [Janssen and Corporaal 1997], and a node-splitting algorithm, called "Controlled Node Splitting", is presented that tries to minimize the number of splits. However, their algorithm differs from our approach in that they use the traditional approach of splitting *one* node while we exclude one node from splitting and split *all other nodes* in the MSED-set. Janssen and Corporaal restrict the set of nodes that are candidates for splitting 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.

The set RC is defined as the set of nodes of SED-sets that dominates other SED-sets and that are reachable from them.

When applying these restrictions, code sizes were reduced to almost one tenth of the original size. Our measurements indicated that these savings were mostly due to the heuristic used for node selection. Restriction one (choose SED nodes only) excludes nodes that are outside the irreducible region but can have different predecessors within this region, *i.e.*, this occurs when different loop exits branch to a common successor. We hardly ever observed such a case since transformations T1/T2 already eliminated most non-SED nodes and for the remaining nodes, heuristic selection seemed sufficient. Restriction two (do not choose RC nodes) only applies when multiple SED-sets exist within an irreducible region, *i.e.*, if multiple loops are contained in the region, each of which comprise independent irreducible regions. We only found one such case in our test set.

On a more abstract level, restriction two implies that a node must not be split while its domain is itself irreducible. Consider Figure 9. Since the nesting is not

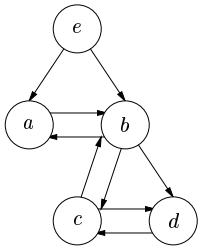


Fig. 9. An Example with an RC-Node. (Node b)

known to the algorithm, it might split node a as the only possibility in the MSED-set $\{a, b\}$, though node b could have been split once c or d had been split and reduced into it. If a is very heavy compared to b, c and d, this might increase the code size and would have been avoided by always reducing the domains first. Hence, the algorithm would never come upon an RC node and would have the freedom to choose b if it is lighter than a, even though it contains c and d. However, this would always result in splits of entire domains, which leads to the problems discussed in the context of Figure 4. Another problem was illustrated in the example of Section 2. Transformations T1 and T2 might reduce parts of the control-flow graph into an MSED-node that are not even part of the loop and, therefore, does not need to be split. This problem has not been solved by the above restrictions.

Partial redundancy elimination [Knoop et al. 1992] can handle code motion for irreducible regions. It supports a different form of strength reduction that restricts the handling of constants to source code constants instead of loop constants. We focus on the recognition and optional transformation of irregular loops that can be utilized for any optimization, such as global register coloring, induction variable elimination and software pipelining.

8. CONCLUSION

We formally defined and showed the correctness of a new approach for optimized node splitting that transforms irreducible regions of control flow into reducible ones. This method is superior to approaches previously published since it reduces the number of replicated nodes by comparison. We also discussed the application of DJ Graphs to recognize the structure of irreducible loops and implemented extensions to common code optimizations to handle these new types of loops. We evaluated the performance of an implementation of both optimized node splitting and optimizations for irregular loop via DJ Graphs in an optimizing compiler backend by comparing them with unoptimized irreducible loops, which is the common approach in contemporary optimizers, as well as traditional node splitting. Our results showed improvements of 1-40% in the number of executed instructions for the approaches of handling irreducible loops. Optimized node splitting has the advantage that it does not require changes to other code optimizations within the compiler but may increase the code size of large programs by about 2% and the size of small programs by about 12%. On the average, it results in less code growth than traditional node splitting and, hence, is superior to it. Recognizing loops via DJ Graphs requires moderate changes to the optimizer and does not significantly change the code size. We suggest the use of DJ Graphs for the design of new compilers and optimized node splitting to retrofit existing compilers as means to optimize irreducible loops. The benefits of handling irregular loops should even be more significant in the context of global instruction scheduling to exploit instruction-level parallelism by the compiler and enhanced modulo scheduling, which may introduce irreducible loops. There issues are becoming increasingly important for contemporary architectures with a VLIW design.

Acknowledgments

The comments by the anonymous reviewers helped to improve this paper. In particular, we like to thank anonymous reviewer 1 for the time and effort he put into providing very detailed feedback on drafts of this paper.

APPENDIX

A. ALGORITHM FOR OPTIMIZED NODE SPLITTING

In the following, the recursive algorithm for T_r is given. Transformations are initiated by a call splt_loops(start node, empty set). The first argument to `splt_loops` is the node dominating all nodes that have yet to be processed. The second argument is a set of nodes that, if non-empty, defines a region that should be handled since all nodes outside of it have already been processed and did not change. The function returns `true` if the given node has any edges that indicate an irreducible loop at its level.

```
bool splt_loops(top, set) {
  cross = false;
  foreach (child in domtree.successors(top))
    if (set is empty OR child in set)
      if (splt_loops(child, set))
        cross = true;
```

```
  if (cross) handle_ir_children(top, set);
  foreach (predecessor in controlflow.preds(top))
    if (is_sp_back(pred, top) AND !(top dominates pred))
      return true;
  return false;
}
```

At first, `splt_loops` handles all levels below the given node (but only within the current region). If any of these calls return `true`, then a child of `top` contains an irreducible loop on the level just below `top`. This is handled in a bottom-up fashion so that the domains in that loop are already reducible.

After the children of the current `top` have been handled completely, it is checked if there exits any edge that indicates an irreducible loop on the level of `top` itself and the result is returned.

The function `handle_ir_children` is called with the external dominator node as an argument. It has to find all SED-maximal loop-sets and then split the irreducible ones one after another.

```
void handle_ir_children(top, set) {
  // find all strongly conntected components (SCCs)
  scclist = find_sccs(child, set, top.level);
  foreach (scc in scclist)
    if (size_list(scc) > 1) // non-trivial component
      handle_scc(top, scc);
}
```

After all SCCs have been found, they are now converted one by one into reducible regions by `handle_scc`.

```
void handle_scc(top, scc) {
  ComputeWeights(top, scc);
  // find header w/ max sum(weights(nodes in domain))
  hdr = ChooseNode(msed);
  // split nodes in scc (except hdr and it domain)
  SplitSCC(hdr, scc);
  RecomputeDJG(top); // renew control-flow and DJ info
  // add copies that are headers to tops
  tops = find_top_nodes(scc);
  foreach (hdr in tops)
    splt_loops(hdr, scc); // recurse: split all headers
}
```

`SplitSCC` then splits all nodes in the SCC except the chosen node and its domain, rearranges the control flow graph and changes the dominator information such that the copied regions are independent subtrees in the dominator tree.

```
void SplitSCC(header, scc) {
  make a copy of nodes in {scc - domain(header)};
  connect copies (within loop set and immediate
    neighbors outside loop set), renew DJ info;
  scc = scc + copies;
}
```

The heuristic selects the node with the maximum weight out of headers in the MSED-set.

```
node ChooseNode(msed) {
  MaxWeight = 0;
  foreach (node in msed)
    if (node.weight > MaxWeight) {
      MaxWeight = node.weight;
      MaxNode = node;
    }
  return MaxNode;
}
```

Weights of header nodes are computed as the sum of the weights of nodes in the domain of the header. The weight of a single node `sigma` is determined as the number of RTLs of this node (instructions in the intermediate representation) excluding branches and jumps.

```
void ComputeWeights(top, scc) {
  foreach (node in scc)
    if (node.level == top.level + 1) {
      GetWeight(node, node, scc);
      add_list(node, msed);
    }
}

void GetWeight(node, header, scc) {
  node.weight = sigma(node);
  foreach (child in domtree.successors(node))
    if (in_list(child, scc)) {
      GetWeight(child, header, scc);
      node.weight = node.weight + child.weight;
    }
  node.header = header;
}
```

See [Unger 1998] for more details including an adapted algorithm for finding strongly connected components (SCCs) [Cormen et al. 1993].

B. PROOF OF CORRECTNESS (CONTINUED)

As mentioned before, the remainder of the proof consists of the following steps: A minimal sequence is constructed by choosing a loop L of flow graph G within some constraints. This L is identified within the minimal, reducible equivalence graph G_{min} to find the header h for the transformation $\mathrm{T}_r(G, L, h)$, which is accomplished by describing equivalent loop sets of G and G_{min} via their labeling. Then it is shown that

—there is always a loop-set L with exactly one equivalent loop-set;

—a header of the graph after applying T_r is always a header node of L; and

—the final (reducible) graph G from this construction is minimal wrt. its weights since $\sigma(G) = \sigma_{min}(G_{min})$.

But first, a number of terms have to be defined.

B.1 Definitions for the Proof

In the following, we assume flow graphs $G = (N, E, s)$ with nodes N, edges E and a start node s whose dead code (unreachable nodes and their edges) have been removed. First, labeling and equivalence properties of flow graphs have to be introduced.

Definition 10 (Labeling). *Let $G = (N, E, s)$ be an arbitrary flow graph. A total function $l : N \to \mathbb{N}$ is then called a* labeling *of G if for all $(n, m) \in E$ and $(n, m') \in E$ the following holds:*

$$l(m) = l(m') \longrightarrow m = m'$$

Note that l is not requested to be injective. Indeed, the case where l is not injective provides the necessary notion of *equivalent* nodes: All nodes that share the same label are equivalent. With that notion we can now define exactly when two flow graphs are equivalent.

Definition 11 (Equivalent flow graphs). *Two flow graphs G_1 and G_2 with labelings l_1 and l_2, respectively, are said to be* equivalent*, in short $G_1 \sim G_2$, if $l_1(s_1) = l_2(s_2)$ and if for every path $p = (p_1, \ldots, p_k)$ of G_1 there is a path $q = (q_1, \ldots, q_k)$ of G_2 such that*

$$\forall 1 \leq i \leq k : l_1(p_i) = l_2(q_i)$$

and vice versa.

In the initial flow graph an injective labeling l is chosen. But if any nodes are duplicated, their labels are kept on all copies.

Theorem 6. *$\sim$ is an equivalence-relation.*

The proof follows directly from the definition. Next, it is shown that Theorem 2 also holds for the e-domain from Definition 7:

Theorem 7. *All edges into* e-domain *originate from e.*

Proof. (indirect)
Let $(m, l) \in E$ be an edge, such that $l \in e$-domain and $e \neq m \notin e$-domain. Then one of the following would have to be true:

—e does not dominate m
But then e does not dominate l, which is a contradiction to $l \in e$-domain.

—$m \in L$
But since $l \in e$-domain, there must be a path p from l into L that does not touch e. This, however, means, that e dominates all nodes in p. Thus, $L \cup p$ would also be a loop-set and MSED-set $(L \cup p) \subseteq$ MSED-set (L). This is not possible if L is SED-maximal.

—There is no path from m into L that does not touch e.
But then, there is no such path originating from l either. This is a contradiction to $l \in e$-domain.

□

B.2 Completing the Proof

Throughout the following proof, several symbols will be used without being defined over and over again. These are:

—$G = (N, E, s)$ stands for the flow graph that is being transformed. For each iteration G is the graph *before* T_r is applied.
—$G' = (N', E', s')$ is the result of the application of T_r in the current iteration.
—$G_{min} = (N_{min}, E_{min}, s_{min})$ is a minimal, reducible graph equivalent to G.
—l, l' (the corresponding labeling) and l_{min} are labelings of G, G' and G_{min}, respectively.
—L is an irreducible, SED-maximal loop-set of G.
—h is a header node of L.
—$S = L \setminus \mathrm{domain}(h)$
—b is a total function $b : N_{min} \to N$, such that $\forall n \in N_{min} : l_{min}(n) = l(b(n))$.
—b' is a total function $b' : N_{min} \to N'$, with $\forall n \in N_{min} : l_{min}(n) = l'(b'(n))$.

These functions b and b' will be used to map each node of G_{min} to the node of G and G', respectively, from which it is a copy.

The proof will construct a minimal sequence by choosing an L of G within some constraints. This L will then be looked up in G_{min} to find the node h for the transformation $\mathrm{T}_r(G, L, h)$. For this look-up it is necessary to define, which loop-set in G_{min} is related to L. This is done in the next definition:

Definition 12 (Equivalent Loop-set). *Let G, G_{min}, L, l, l_{min} and b be defined as above.*

Then a loop-set L_{min} of G_{min} is called an equivalent loop-set of L, if and only if L_{min} is maximal such that

$$b(L_{min}) = L$$

Note that the L_{min} is not necessarily unique. However, if b is constructed as below, then there is always at least one equivalent loop-set.

For the initial graph, where l is an injective labeling of G, b is constructed as

$$\forall n \in N_{min} : b(n) = l^{-1}(l_{min}(n)).$$

Then, in each iteration, L is selected such that there is exactly one equivalent loop-set L_{min} of L. It will be shown later that such an L does always exist.

Since L_{min} is a loop-set of G_{min}, which is reducible, it must be a reducible loop-set itself and thus have a single entry node h'. Then, $h = b(h')$ is one of the header nodes of L and, therefore, the graph G' can be constructed by $G' = \mathrm{T}_r(G, L, h)$. The labeling and weight of G' has already been defined. The function $b' : N_{min} \to N'$ is defined as

$$\forall n \in N_{min} : b'(n) = \begin{cases} (b(n), 2) & \text{If } n \in L_{min} \wedge b(n) \in S \\ (b(n), 1) & \text{otherwise} \end{cases}$$

Thus, a new iteration can be done with G', l', σ' and b' as G, l, σ and b, respectively, if G' is still irreducible.

For the above algorithm to be correctly defined, the following three things have to be shown:

—There is always a loop-set L with exactly one equivalent loop-set. This is proved in Theorem 8.

—$b(h')$ is always a header node of L, which is proved in Theorem 9.

—For the finally constructed (reducible) graph G: $\sigma(G) = \sigma_{min}(G_{min})$, which is shown in Theorem 10.

The last point is not strictly needed for the above algorithm but it will show that the constructed graph is minimal.

To be able to prove the above items, several lemmas will be needed.

Lemma 1 (s is unique). *Let G, G_{min}, l, l_{min} and b be defined as above. Then*

(i) $\nexists n \in N_{min} \setminus \{s_{min}\} : l_{min}(n) = l_{min}(s_{min})$ *and*

(ii) $\nexists n \in N \setminus \{s\} : l(n) = l(s)$.

PROOF. (i) Since s_{min} dominates all nodes of G_{min}, any edge $(u, n) \in E_{min}$ where $l_{min}(n) = l_{min}(s_{min})$ can be replaced by the edge (u, s_{min}) without destroying the reducibility or equivalence of G_{min}. But then n has no longer any incoming edge and can be removed from G_{min} thus making G_{min} lighter. This is a contradiction to the assumption, that G_{min} is minimal. Therefore, such a node n cannot exist.

(ii) This is obviously true for the initial graph G with an injective labeling. Since s dominates all nodes in G, it cannot be in any irreducible loop and, therefore, cannot be split by the algorithm. Thus, the above property holds through all iterations of the algorithm.

□

Flow graphs, which are equivalent to a graph with an injective labeling have a special property, which is much stronger than simple equivalence as defined in Definition 11.

Lemma 2. *Let G, G', G_0 be flow graphs with labelings l, l' and l_0, respectively, such that $G \sim G' \sim G_0$. Furthermore let l_0 be injective and let G and G' be such that the property of Lemma 1 holds. Then the following is true:*

For all $q_0 \in N'$ and for each path $p = (p_0, \ldots, p_k)$ of G with $l'(q_0) = l(p_0)$ exists exactly one path $q = (q_0, \ldots, q_k)$ of G' with $l'(q_i) = l(p_i)$

PROOF. Since G' was supposed to not contain dead code, there is a path $q' = (q'_{-j}, \ldots, q'_0)$ of G' with $q'_{-j} = s'$ and $q'_0 = q_0$. Since $G' \sim G_0$ there must be a path $\overline{q} = (\overline{q}_{-j}, \ldots, \overline{q}_0)$ of G_0 with $l_0(\overline{q}_i) = l'(q'_i)\ \forall -j \leq i \leq 0$. Since l_0 is injective $\overline{q}_{-j} = s_0$. Since $G \sim G_0$ there must be a path $\overline{p} = (\overline{p}_0, \ldots, \overline{p}_k)$ of G_0 with $l_0(\overline{p}_i) = l(p_i)\ \forall 0 \leq i \leq k$. Furthermore, since l_0 is injective there can only be one such path $\overline{p}$ and $\overline{p}_0 = \overline{q}_0$. Thus, $\overline{qp} = (\overline{q}_{-j}, \ldots, \overline{q}_0, \overline{p}_1, \ldots, p_k)$ is a path of G_0 and because $G_0 \sim G'$ there must be a path $r = (r_{-j}, \ldots, r_0, \ldots, r_k)$ of G' such that $l'(r_i) = l_0((\overline{qp})_i)\ \forall -j \leq i \leq k$. All that remains to be shown is that $r_0 = q_0$ and thus that $q = (r_0, \ldots, r_k)$ is a path as requested.

From $l_0(s_0) = l_0((\overline{qp})_{-j}) = l'(r_{-j}) = l(s')$ and from the property of Lemma 1 follows that $r_{-j} = s' = q'_{-j}$. From Definition 10 follows that $r_i = q'_i \longrightarrow r_{i+1} = q'_{i+1}$ and thus (by induction) that $r_0 = q'_0 = q_0$. From Definition 10 also follows that q is the only path originating from q_0 following the same labels as p. □

Lemma 3. *Let G, G_{min}, l, l_{min} and b be defined as above.*
Then, for every path $q = (q_1, \ldots, q_k)$ of G_{min}, $(b(q_1), \ldots, b(q_k))$ is a path of G.

PROOF. From Lemma 2 and Lemma 1 follows that there is exactly one path $p = (p_1, \ldots, p_k)$ of G such that $p_1 = b(q_1)$ and $l(p_i) = l_{min}(q_i)$. From $\forall n \in N_{min}$: $l(b(n)) = l_{min}(n)$ and Definition 10 follows that $p_i = b(q_i)$. □

Lemma 4 (b is surjective). *Let G, G_{min}, l, l_{min} and b be defined as above.*
Then b is surjective.

PROOF. G was supposed to not contain dead code
$\Rightarrow \forall n \in N$ there is a path $p = (p_1, \ldots, p_k)$ of G with $p_1 = s$ and $p_k = n$. Since $G \sim G_{min}$ and Lemma 2 there is exactly one path $q = (q_1, \ldots, q_k)$ of G_{min} with $q_1 = s_{min}$ and $l(p_i) = l_{min}(q_i)$. Thus, $b(q_1) = p_1$ and from Definition 10 follows that $b(q_i) = p_i$ and thus that $b(q_k) = n$ □

.

Lemma 5. *Let G be an arbitrary control flow graph. Any node $x \in N$ is outside of all irreducible loops if and only if the following holds:*

$\forall y \in N$ with x is reachable from y and y is reachable from x: $\exists z \in N$ such that z dominates both x and y and z either occurs on every path from x to y or on every path from y to x (or both).

Furthermore, if x is not in any irreducible loop, then the above nodes z are in no irreducible loop either.

The proof uses the following definition that determines if a node is inside of an irreducible loop:

x is inside of an irreducible loop if and only if it is inside a loop-set whose MSED-set contains more than one node.

PROOF. $\Longrightarrow$
Since x and y are reachable from each other, there is at least one loop-set that contains both nodes. Let L be the smallest SED-maximal loop-set with $x, y \in L$. This L is well defined because of Theorem 3. Since $x \in L$, L must be reducible and thus have a single entry node z with z dominates all nodes in L. If there is a path from x to y that does not touch z, then all nodes on this path are part of L (because z dominates all these nodes and L is SED-maximal). The same is true for any path from y to x that does not touch z. However, if two such paths really existed, they would form a loop-set L' that does not include z and with $L' \subset L$. This, in turn, means that there is another SED-maximal loop-set, which is strictly smaller (with respect to inclusion) than L and contains both x and y. This is a contradiction to the assumption that L is the smallest SED-maximal loop-set.

Furthermore, from Theorem 3 and the minimality of L follows that for each loop-set L' with $z \in L'$: $L \subseteq L'$ and thus $x \in L'$ and thus that L' is reducible. Therefore, z is not part of any irreducible loop either.

$\Longleftarrow$ (the negation)
If x is inside an irreducible loop L it must be in domain(h) for some $h \in$ MSED-set(L). Because L is irreducible, there is a node $y \in$ MSED-set(L) with $y \neq h$. Because $x, y \in L$, x is reachable from y and y is reachable from x without leaving L. However, since $\forall z \in L, z \neq y$: z does not dominate y and since y does not dominate x the following is true:
$\forall z' \in N$ with z' dominates x and y: $z' \notin L$ and thus there are paths from x to y and from y to x that do not touch z'. □

Lemma 6 (b is injective). *Let G, G_{min}, l, l_{min} and b be defined as above. Then b is injective outside of irreducible loops. This means that the following is true:*
$\forall m, n \in N_{min}$ with $b(m) = b(n)$ and $b(m)$ is not inside any irreducible loop: $m = n$.

PROOF. Induction over the depth of $b(m)$ in the dominator-tree.
Induction-base:
From Lemma 1 follows directly that the above lemma is true for $m = s_{min}$.
Induction-step:
Indirect: Assume there exist nodes $m, n_0 \in N_{min}$ such that $b(m) = b(n_0)$ and $b(m)$ is not inside any irreducible loop but $m \neq n_0$. However, it is assumed, that the lemma is true for all such nodes m' and n' where $b(m')$ is an ancestor of $b(m)$ in the dominator-tree of G.

The following proof will show that, if such m and n_0 exist, then G_{min} cannot be minimal because it is possible to construct another reducible, equivalent flow graph, which is lighter with respect to σ:

Let m and n_0 be as assumed above. Then the graph G'_{min} is constructed as follows:

— $s'_{min} = s_{min}$

— $E'_{min} = E_{min} \setminus \left\{ (\cdot, n) \in E_{min} \middle| \begin{array}{l} b(n) = b(m) \wedge \\ n \neq m \end{array} \right\}$

$$\cup \left\{ (n_1, m) \middle| \exists (n_1, n_2) \in E_{min} : \begin{array}{l} b(n_2) = b(m) \wedge \\ n_2 \neq m \end{array} \right\}$$

— $N'_{min} = N_{min} \setminus \left\{ n \in N_{min} \middle| \begin{array}{l} \nexists \text{ a path } p = (p_1, \ldots, p_k) \text{ in} \\ E'_{min} : p_1 = s_{min} \wedge p_k = n \end{array} \right\}$

In short, the above construction replaces all edges to any node n with $b(n) = b(m)$ except to m itself with appropriate edges to m. After that the other copies of m are dead-code and can be removed. Since the removed edges have been replaced by edges to a node with the same label and the removed nodes are just those that are no longer reachable, the new graph is equivalent to the old one, i.e. $G'_{min} \sim G_{min}$.

Since at least the node n_0 has been removed from N'_{min}, the above control flow graph is equivalent to and lighter than G_{min}. This would only be possible if it were not reducible since G_{min} was assumed to be minimal. Therefore, it is sufficient to prove that G'_{min} is reducible in order to show that such m and n_0 cannot exist.

However, if G'_{min} is compared to G_{min}, only a few nodes and edges have been removed and some other edges have been added. But neither the removed edges nor the removed nodes may have introduced any irreducibility. Therefore, if G'_{min} were irreducible, this irreducibility would have been introduced by the *added* edges. These, however, are all targeted at m. Therefore, if G'_{min} is irreducible, m must be part of at least one of the irreducible loops.

Thus, it is sufficient to show that m is outside of all irreducible loops. By Lemma 5 this can be shown by proving that:

$\forall y \in N$ with m is reachable from y and y is reachable from m: $\exists z \in N$ such that z dominates both m and y and z either occurs on every path from m to y or on every path from y to m (or both).

Let $y \in N'_{min}$ be such that y is reachable from m and m is reachable from y in G'_{min}. By the construction of G'_{min} these two nodes must also be reachable from each other in G_{min} and thus by Lemma 3 $b(m)$ is reachable from $b(y)$ and vice versa. Since $b(m)$ is not in any irreducible loop (consider the prerequisites of the lemma) and because b is surjective according to Lemma 4 there is a node $z \in N_{min}$ with $b(z)$ dominates $b(m)$ and $b(y)$, and $b(z)$ either occurs on every path from $b(m)$ to $b(y)$ or on every path from $b(y)$ to $b(m)$ (or both).

Assume first that $b(z) \neq b(m)$.
Then $b\,(z)$ is an ancestor of $b\,(m)$ in the dominator-tree and by Lemma 5 not in any irreducible loop either. Therefore, b is injective at $b(z)$ and thus $\nexists z' \in N_{min}$ with $b(z') = b(z)$. From this and Lemma 3 follows that z dominates y and all nodes $n \in N_{min}$ with $b(n) = b(m)$. Because $b(z) \neq b(m)$ and the above, $z \in N'_{min}$ and z dominates m in G'_{min}. This, however, means that the added edges to m cannot influence the dominance of z over y. Thus, z dominates both m and y in G'_{min}.

By Lemma 3 and the uniqueness of z, if $b(z)$ occurred on every path from $b(m)$ to $b(y)$, then z occurs on every path from m to y in G_{min} and thus on every path from m to y in G'_{min}. On the other hand, if $b(z)$ occurred on every path from $b(y)$ to $b(m)$, then, again by Lemma 3 and the uniqueness of z, z occurs on every path of G_{min} from y to any node n with $b(n) = b(m)$ and thus it occurs on every path of G'_{min} from y to m. Thus, m is not in any irreducible loop in G'_{min} and therefore, G'_{min} is completely reducible.

On the other hand, if $b(z) = b(m)$, then $b(m)$ dominates $b(y)$. Thus, by Lemma 3, on every path of G_{min} from s_{min} to y a node $n \in N_{min}$ with $b(n) = b(m)$ must occur. This, in turn, means that m must occur on every path of G'_{min} from s'_{min} to y. Thus, m dominates y and occurs on every path from m to y and vice versa. Therefore, m cannot be in any irreducible loop and thus G'_{min} must be completely reducible.

As was said above, this is not possible if G_{min} was truly minimal and thus such nodes m and n_0 cannot exist. □

Lemma 7 (Existence of equivalent loop-sets). *Let G, G_{min}, l, l_{min} and b be defined as above.*
Then for each loop-set L of G, there is an equivalent loop-set L_{min} of G_{min}.

PROOF. Since G was supposed to not contain dead code and since L is a loop-set there is at least one infinitely long path $p = (p_1, p_2, \ldots)$ of G such that $p_1 = s$ and $\exists k \forall j > k : p_j \in L$ and $\forall k \forall n \in L \exists j > k : p_j = n$. From Lemma 2 follows that there

is a path $q = (q_1, q_2, \ldots)$ of G_{min} such that $q_1 = s_{min}$ and $l(p_i) = l_{min}(q_i)$. Since G_{min} is finite there must be a loop-set L_{min} such that $\exists k' \forall j > k' : q_j \in L_{min}$ and $\forall k \forall n \in L_{min} \exists j > k : q_j = n$. From Lemma 3 follows that $b(q) = (b(q_1), b(q_2), \ldots)$ is a path of G following the same labels as p. Since, furthermore, $b(q_1) = s = p_1$ follows from Definition 10 that $b(q) = p$ and thus that $b(L_{min}) = L$. □

With the above lemmas the remaining properties as listed on page 26 can be proven.

Theorem 8 (Uniqueness of equivalent loop-sets). *Let G, G_{min}, l, l_{min} and b be defined as above.*
Then, there is always at least one loop-set L of G, which has exactly one equivalent loop-set L_{min}.

PROOF. Let L be a loop-set of G such that the external dominator e of L is not in any irreducible loop. Since s is never in any irreducible loop and because of Theorem 3 and because G is finite, such a loop-set must always exist. By Lemma 7, L must have at least one equivalent loop-set. Thus, it is enough to show that L has at most one equivalent loop-set.

This is shown indirectly: Assume there are two different equivalent loop-sets $L_{min,1}$ and $L_{min,2}$. Because equivalent loop-sets are maximal, either on every path from $L_{min,1}$ to $L_{min,2}$ or on every path from $L_{min,2}$ to $L_{min,1}$ a node n with $b(n) \notin L$ must occur. Without restricting generality, it is assumed that it occurs on every path from $L_{min,1}$ to $L_{min,2}$. Let m be the header-node of $L_{min,1}$. Then another flow graph G'_{min} can be constructed as in the proof for Lemma 6, except that only edges to nodes n with $b(n) = b(m)$ and $n \in L_{min,2}$ are replaced. Then, for the same reasons, $G'_{min} \sim G_{min}$ and since $b(L_{min,2}) = L$ at least one such node n has been removed. Therefore, it is again enough to show that G'_{min} is reducible in order to prove that such two equivalent loop-sets cannot exist.

Since only edges to m have been added, the above assumption about paths from $L_{min,1}$ to $L_{min,2}$ is still true in G'_{min}.

As in the proof of Lemma 6, if G'_{min} is irreducible, this irreducibility could only have been introduced by the added edges to m. Thus, m would have to be in at least one irreducible loop-set. It is therefore enough to show that

$\forall y \in N'_{min}$ with m and y are reachable from each other: $\exists z \in N'_{min}$ such that z dominates y and m and z either occurs on every path from m to y or on every path from y to m.

Let y be an arbitrary node in G'_{min} such that y and m are reachable from each other. If $y \in L_{min,1}$, then m dominates y and thus $z = m$ is such a node. Therefore, $y \notin L_{min,1}$. Then on every path $p = (m, \ldots, y, \ldots, m)$ a node n with $b(n) \notin L$ must occur, since $L_{min,1}$ is maximal. Thus, there is an edge (p_i, p_{i+1}) in p such that $b(p_i) \in L$ and $b(p_{i+1}) \notin L$. From Theorem 7 follows that $b(p_{i+1}) \notin e$-domain. This means that $\exists j > i : b(p_j) = e$. However, since e is not in any irreducible loop, there is only one node e_{min} with $b(e_{min}) = e$ and thus e_{min} occurs on every path $p = (m, \ldots, y, \ldots, m)$.

This, however, means that for each loop-set L' of G'_{min} with $m, y \in L'$: $e_{min} \in L'$. Let L'_0 be the smallest such loop-set with respect to inclusion. This is well defined because of Theorem 3. From Lemma 3 follows, that $b(L'_0)$ is a loop-set

and $e \in b(L'_0)$. Thus, $b(L'_0)$ is reducible and has a single header-node. Since b is surjective, there is a node $z \in N_{min}$ such that $b(z)$ is that header-node and thus dominates all nodes in $b(L'_0)$. Furthermore, because $b(z) \notin L$, $z \in N'_{min}$. Because of Theorem 3 $b(z)$ is not in any irreducible loop and thus z is unique. This, however, means that z dominates all nodes in L'_0 (including y and m) and thus that L'_0 is reducible and z is its single entry-node. Then, however, z must either occur on every path from m to y or on every path from y to m because L'_0 was the smallest loop-set containing y and m.

Thus, m cannot be in any irreducible loop and G'_{min} is completely reducible. This is not possible if G_{min} is minimal as assumed and thus L cannot have two (or more) equivalent loop-sets. □

Theorem 9 ($b(h')$ is a header node). *Let G, G_{min}, l, l_{min} and b be defined as above.*
Let L be a SED-maximal loop-set of G, e its external dominator and e not in any irreducible loop-set. Let L_{min} be the equivalent loop-set of L and h' the single entry-node of L_{min}.
Then $b(h') \in$ MSED-set(L).

Proof. Indirect.
Let L be such a loop-set, L_{min} its equivalent loop-set and h' the entry node of L_{min} with $b(h') \notin$ MSED-set(L). Then by Theorem 1 $b(h') \in$ domain(h) for some $h \in$ MSED-set(L). Because L_{min} is an equivalent loop-set, there is a node $m \in L_{min}$ with $b(m) = h$. On the other hand, since h dominates $b(h')$ in G, on every path from s_{min} to h' a node n with $b(n) = h$ must occur. Furthermore, because h' is the entry node of L_{min}, $n \notin L_{min}$ and thus $n \neq m$. Thus, if G'_{min} is constructed as in the proof for Lemma 6, then $G'_{min} \sim G_{min}$, for the same reasons as given there, and $\sigma(G'_{min}) < \sigma(G_{min})$ since at least n has been removed. Thus, it is sufficient to show that G'_{min} is reducible in order to prove that h' must be a header node of L.

If G'_{min} was reducible, m had to be in at least one irreducible loop, since only edges to m have been added. Thus, all that remains to be shown is that:

$\forall y \in N'_{min}$ with m and y are reachable from each other: $\exists z \in N'_{min}$ such that z dominates y and m, and z either occurs on every path from m to y or on every path from y to m.

Let y be such a node.
Since $h = b(m)$ dominates $b(h')$ in G and since h' dominates all nodes in L_{min}, on every path of G_{min} from s_{min} to any node in L_{min} at least one node n with $b(n) = h$ must occur. Since all added edges are edges to m and $b(m) = h$, this must still be true for G'_{min}. Thus: On every path of G'_{min} from s'_{min} to any node in $L_{min} \cap N'_{min}$ at least one node n with $b(n) = h$ must occur. However, by the construction of G'_{min}, m is the *only* node in N'_{min} with $b(m) = h$ and thus m dominates all nodes of $L_{min} \cap N'_{min}$. Therefore, if $y \in L_{min}$, then $z = m$ fulfills the above requirements.

Thus, let $y \notin L_{min}$.
Since $b(y) \in L$ and L_{min} is the only equivalent loop-set of L, there must be a path q of G_{min} from y into L_{min} such that $b(q_i) \in L$ for all i. However, since $y \notin L_{min}$, this means that on every path of G_{min} from any node in L_{min} to y, a node n with

$b(n) \notin L$ must occur. In particular, this is true for m and since only edges to m have been added in G'_{min} it is also true that: On every path $p = (m, \ldots, y)$ of G'_{min} a node n with $b(n) \notin L$ must occur. Since the same applies to every path $p = (m, \ldots, y, \ldots, m)$ the arguments of the proof of Theorem 8 apply here as well and there is a node z, which fulfills the above requirements.

Therefore, m cannot be in any irreducible loop and G'_{min} must be completely reducible. Since this is a contradiction to G_{min} minimal, h' must be a header node of L. □

Theorem 10 (The constructed graph is minimal). *Let G, G_{min}, l, l_{min} and b be defined as above. Furthermore, let G be the final graph, namely let G be reducible. Then $\sigma(G) = \sigma_{min}(G_{min})$.*

PROOF. By Lemma 4 b is surjective.
Since G is completely reducible, b is completely injective by Lemma 6.
Thus, b is bijective and thus $\sigma(G) = \sigma_{min}(G_{min})$. □

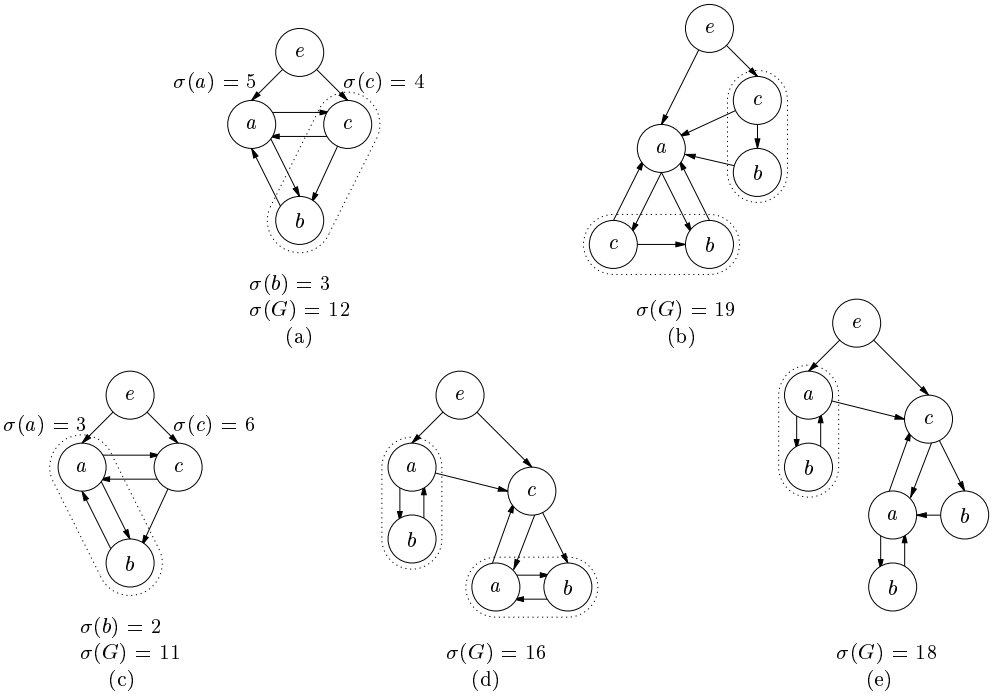
The previous theorems show that there is always a sequence that constructs a minimal, equivalent and reducible flow graph for any control flow graph G. The proof, however, already used such a minimal graph for the selection of the header node, on which the transformation T_r is applied and, therefore, does not provide a usable method for choosing the nodes. The Theorems 4 and 5, on the other hand, were proven independently of any order, and, therefore, if another selection scheme is used, the algorithm remains correct, though possibly not minimal.

In the proof above, it was necessary to handle the *outer* loops first. This, however, is not really necessary. It is possible to give a similar (but even more technical and incomprehensible) proof that does not depend on any order in which the irreducible loop-sets L are chosen.

Figure 10 shows the steps of the algorithm for two different selections of the h_i. Any of the resulting flow graphs may be minimal depending on the weight of the nodes. Table 10(f) lists different weights and the corresponding minimal graph.

As can be seen from this table, the weight of the nodes alone is not sufficient to decide which node has to become h_i in each step. The resulting number of copies for other nodes has to be considered as well. Though it might be possible to deduce that number from the structure of the MSED-set without actually constructing each possible final graph, such a method could not be found.

Still, the weight of the nodes is known from the beginning and can be used as a heuristic for the selection of the h_i. The algorithm outlined in Section 4.1 and in Appendix A uses such a heuristic by choosing the header h_i such that $\sigma(\text{domain}(h_i))$ is maximal for all possible h_i. This, however, requires that the final weight of the domains is known and thus that the inner (nested) loops are converted first.



Weight of			Minimal		
a	b	c	sequence	graph	weight
5	3	4	a	10(b)	19
4	3	5	a	10(b)	20
3	2	6	c, a	10(e)	18

(f) Weights for graphs

Fig. 10. Two possible traces of the recursive alg.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley.

Allen, F. 1970. Control flow analysis. *Sigplan Notices 5,* 7, 1–19.

Bala, V., Duesterwald, E., and Banerjia, S. 2000. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 1–12.

Benitez, M. E. 1991. Register transfer standard. TR RM-91-01, University of Virginia. Mar.

Benitez, M. E. and Davidson, J. W. 1988. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 329–338.

Bilardi, G. and Pingali, K. 1996. A framework for generalized control dependence. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.* Philadelphia, Pennsylvania, 291–300.

Cocke, J. and Miller, J. 1969. Some analysis techniques for optimizing computer programs. In *2nd Hawaii Conference on System Sciences.* 143–146.

Colwell, R. P., Nix, R. P., O'Donnel, J. J., Papworth, D. B., and Rodman, P. K. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers 37,* 8 (Aug.), 318–328.

Cormen, T., Leiserson, C., and Charles, E. 1993. *Introduction to Algorithms.* MIT Press.

DAVIDSON, J. W. AND WHALLEY, D. B. 1990. Ease: An environment for architecture study and experimentation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 259–260.

FISCHER, C. N. AND LEBLANC, R. J. 1991. *Crafting a Compiler with C*. Benjamin Cummings.

HAVLAK, P. 1997. Nesting if reducible and irreducible loops. *ACM Trans. Programming Languages and Systems 19,* 4 (July), 557–567.

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

HENNESSY, J. AND PATTERSON, D. 1996. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann.

HOOGERBRUGGE, J. AND AUGUSTEIJN, L. 1999. Instruction scheduling for trimedia. *Journal of Instruction-Level Parallelsim 1,* 1-2. www.jilp.org.

JANSSEN, J. AND CORPORAAL, H. 1997. Making graphs reducible with controlled node splitting. *ACM Trans. Programming Languages and Systems 19,* 6 (Nov.), 1031–1052.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 224–233.

MUCHNICK, S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

MUELLER, F. 1991. Avoiding unconditional jumps by code replication. M.S. thesis, Dept. of CS, Florida State University.

MUELLER, F. AND WHALLEY, D. B. 1992. Avoiding unconditional jumps by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 322–330.

MUELLER, F. AND WHALLEY, D. B. 1995. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 56–66.

PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 16–27.

RAMALINGAM, G. 1999. Identifying loops in almost linear time. *ACM Trans. Programming Languages and Systems 21,* 2 (Mar.), 175–188.

RAMALINGAM, G. 2000. On loop, dominators, and dominance frontier. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 233–241.

SREEDHAR, V., GAO, G., AND LEE, Y. 1996a. Identifying loops using DJ graphs. *ACM Trans. Programming Languages and Systems 18,* 6 (Nov.), 649–658.

SREEDHAR, V., GAO, G., AND LEE, Y. 1996b. A new framework for exhaustive and incremental data flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–290.

UNGER, S. 1998. Transforming irreducible regions of control flow into reducible regions by optimized node splitting. M.S. thesis, Humboldt University Berlin, Germany. Studienarbeit, http://www.informatik.hu-berlin.de/~mueller/ftp/pub/mueller/theses/unger-st.ps.gz.

UNGER, S. AND MUELLER, F. 2001. Handling irreducible loops: Optimized node splitting vs. dj-graphs. TR 146, Inst. f. Informatik, Humbolt University Berlin. Jan. www.informatik.hu-berlin.de/~mueller.

WARTER, N. J., HAAB, G. E., SUBRAMANIAN, K., AND BOCKHAUS, J. W. 1992. Enhanced modulo scheduling for loops with conditional branches. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*. 170–179.

WEAVER, D. L. AND GERMOND, T. 1994. *The SPARC Architecture Manual – Version 9*. Prentice Hall.

Received March 2001; revised December 2001; accepted March 2002