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

Sebastian Unger
Digital Media Systems GmbH
Otto-Schmirgal-Str. 3
10319 Berlin (Germany)

Frank Mueller
Humboldt University Berlin, Institut f. Informatik
10099 Berlin (Germany)
mueller@informatik.hu-berlin.de

ABSTRACT

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.

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 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 [13]. On one side, global instruction scheduling 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. Other code optimization techniques, such as trace scheduling and profile-guided code positioning for optimizing compilers [7, 9, 20], may be combined with code replication [18, 19] or applied during binary translation [3], which can result in irreducible loops.

Common algorithms to detect loops only cover natural loops [1]. 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 [1]. 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 implementation and benefit in performance. It also contributes methods to extend loop optimizations for the second approach when irreducible loops are recognized.

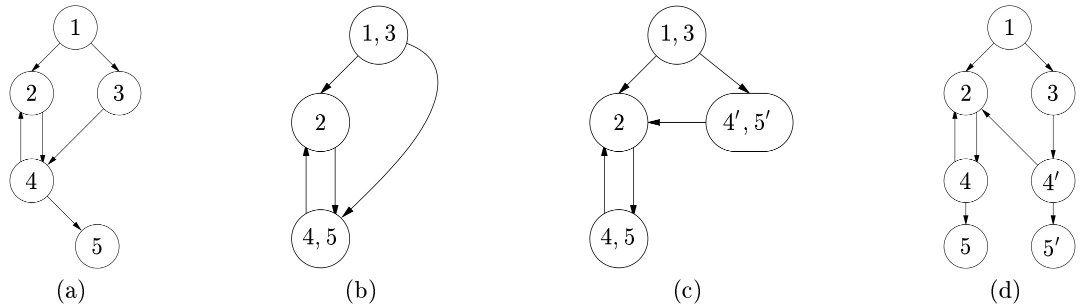


Figure 1: Process of Node Splitting

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 [1]. 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 [11] 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 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 Janssen and Corporaal [14] who found 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:*

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

For a definition of dominators, immediate dominators and other techniques for control flow analysis see, for instance, Muchnick [16].

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

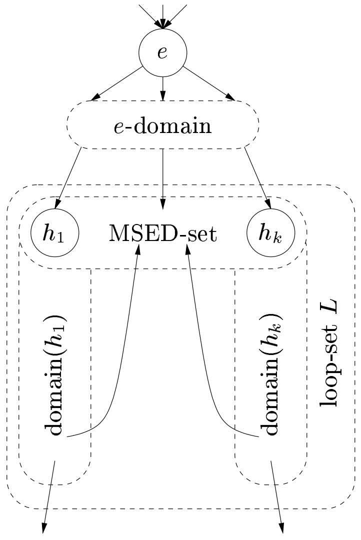


Figure 2: Structure of Irreducible Loops

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:*

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

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)*

The proofs of these theorems can be found in [25]. The results are used in the following section to develop an optimized algorithm for node splitting.

4. OPTIMIZED NODE SPLITTING

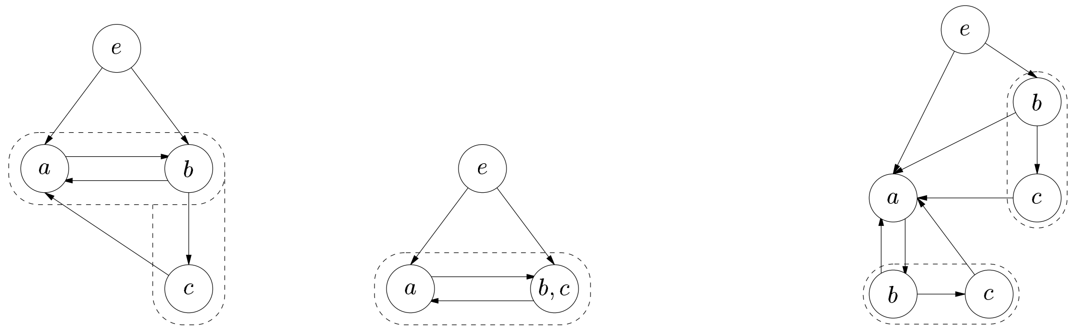
The knowledge about the structure of irreducible loops can be used to guide the T3 transformation to some extend. Repeated application of T1/T2 will collapse domains into their headers leaving an MSED-set. Applying T3 to a node in the MSED-set then splits a header and its *entire* domain (see Figure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

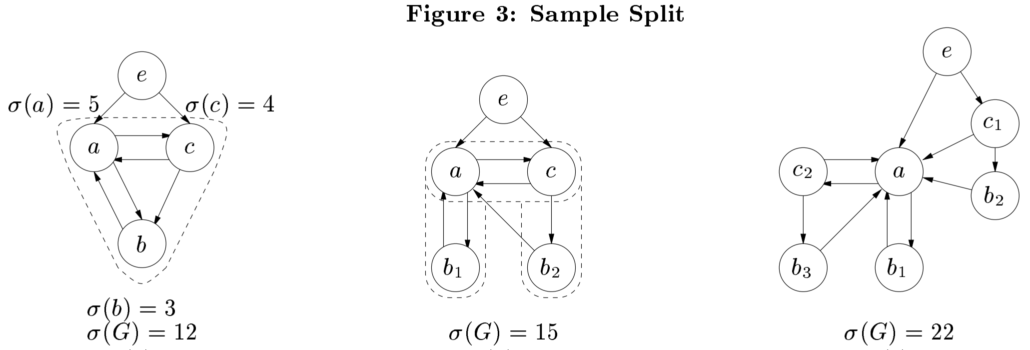
$$e\text{-domain} = \left\{ n_i \in N \;\middle|\; \begin{array}{l} e \textit{ dominates } n_i,\ n_i \notin L \textit{ and} \\ \exists \textit{ a path } p \textit{ from } n_i \textit{ into } L \textit{ with } e \notin p. \end{array} \right\}$$

Does this algorithm always produce the minimal reducible equivalent flow graph as in the example? Unfortunately not, as is shown in Figure 5. Node b_3 could actually be avoided. Its predecessor node c_2 could as well jump to the copy b_1 as shown in Figure 4(c). Had not b but c been split first, Figure 4(c) would have been the result. This means that selecting the nodes to split just by their weight is not sufficient. Another question is if there is always an order that leads to the minimum. Alas, not even that is true. Figure 6 gives



(a) (b) after T2 (c) after T3

Figure 3: Sample Split

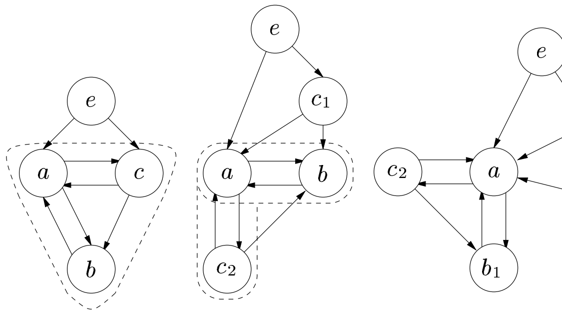


(a) (b) (c)

Figure 5: Splitting Lightest Nodes First (Nodes with same labels have indices for better reference)

a flow graph together with its minimal reducible equivalent graph 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.



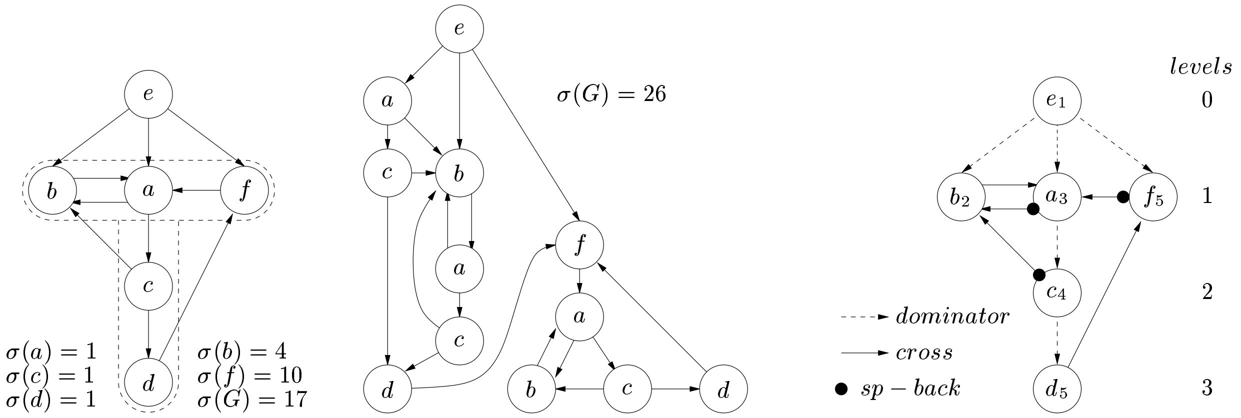
(a) (b) (c)

Figure 4: Best Splitting Sequence (c, b)

First, however, the algorithm is defined more precisely and it is shown that it is correct, i.e., it always produces an equivalent and reducible flow graph. The following shortcut will be used often in the theorems and proofs: 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 $\{f(x)|x \in X\}$.

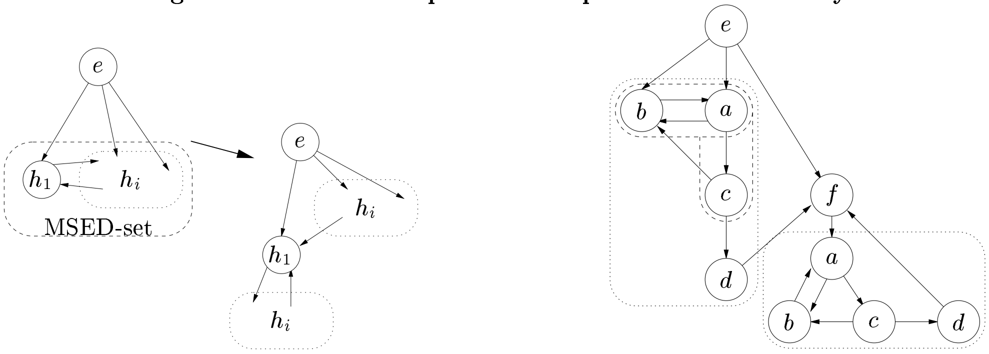
Definition 8 (Transformation T_r). *Let $G = (N, E, s)$ be an arbitrary (irreducible) control flow graph, L an SED-maximal, irreducible loop-set of G, K its MSED-set, e the external dominator and h an arbitrary node from K. Then the transformation $G' = (N', E', s') = T_r(G, L, h)$ is defined as follows (with $S = (L \setminus \mathrm{domain}(h))$):*

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



(a) Original Flow Graph (b) Equiv. Graph (minimal+reducible) (c) DJ-Graph for 6(a)

Figure 6: T3 Cannot Split this Graph in a Minimal Way.



(a) (b)

Figure 7: One Step of the Recursive Algorithm

- $E' \subset N' \times N'$ *such that the following restrictions hold:*
  - $(x, y) \in E \wedge (x, y) \notin (\text{domain}(h) \times S) \iff ((x, 1), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (\text{domain}(h) \times S) \iff ((x, 1), (y, 2)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times (N \setminus S)) \iff ((x, 2), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times S) \iff ((x, 2), (y, 2)) \in E'$
- $s' = (s, 1)$

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). 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 $\mathrm{T}_r(G, L, h)$. 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 $\forall 1 \le i \le n : l(p_i) = l_r(q_i)$ 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 $L \subset \mathrm{domain}(h')$ for some $h' \in$ 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 $L' \subset \mathrm{domain}(h')$ for some $h' \in$ 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 $S \times \{2\} \cup \mathrm{domain}(h) \times \{1\}$ and there is no edge from $S \times \{2\} \cup \mathrm{domain}(h) \times \{1\}$ 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.

Due to space limitations, only a sketch of this proof can be given here (see [25] for the formal proof). 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 $\sigma(G) = \sigma_{min}(G_{min})$.

5. USING DJ GRAPHS TO OPTIMIZE IRREDUCIBLE LOOPS

The representation of DJ Graphs [23] 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 can be detected, including their nesting hierarchy.

An example is depicted in Figure 6(c), which represents the DJ-Graph of the control-flow graph in Figure 6(c). 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 $x \to y$ where $x = y$ or y is an ancestor of x is 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 strongly connected components comprises 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 $d \to c$ in Figure 6(c), 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 [1] 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 [5] 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.

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 [5] as a platform to conduct a performance evaluation. VPO only recognizes natural loops with a single header that the loop is entered through, 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. 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 automata representing an irreducible loop containing two independent natural loops (see Fig. 2a in [23]). Arraymerge merges two sorted arrays into a third sorted one and was originally translated from Fortran. 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) [8] that is integrated into VPO. Table 1 shows in column 1 the name of a program with an irreducible region in some function. (See Table 3 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 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

	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 1: Executed Instructions and their Changes for Irreducible Regions

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 the 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 [26]. 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 2 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 for

	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 2: Size of Irreducible Regions in Instructions and their Changes

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 3 depicting the number of copied register transfer lists (RTLs) [4] for traditional T3 splitting (Column 2) and optimized T_r splitting (Column 3) with changes relative to Column 2 in parenthesis. Since

	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 3: Number of Copied RTLs (Instructions) during Node Splitting

both node 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 3 are not as significant as the differences between T3 and T_r in Table 2. 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 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) with heuristic by Janssen and Corporaal [14]. 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.

Finally, 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 [17], where a miss accounts for 10 cycles delay (for going to the next memory level [12]) 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.

7. RELATED WORK

Reducible flow graphs were first mentioned by Allen [2]. The idea of node splitting stems from Cocke and Miller [6]. DJ Graphs are due to Sreedhar *et. al.* [23]. Loop analysis via DJ Graphs 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.* [24].

Even more recently, Havlak proposed a method for recognizing reducible and irreducible loops as well as the nesting of either ones [10]. 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 [21, 22] 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.

The notion of MSED-sets is introduced by Janssen and Corporaal [14], 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 an irreducible region by itself. We only found one case where multiple loops within an irreducible region occurred.

On a more abstract level, restriction two implies that a node must not be split while its domain is itself irreducible. Consider Figure 8. Since the nesting is not known to the

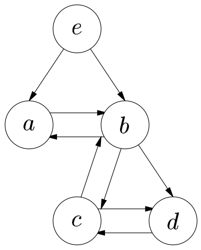


Figure 8: An Example with an RC-Node. (Node b)

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 5. Another problem has 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 [15] can handle code motion for irreducible regions but will support a weaker form of strength reduction since loop constants cannot be recognized [16]. 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 global instruction scheduling.

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 back-end 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, which is becoming increasingly important for contemporary architectures with a VLIW design.

9. REFERENCES

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

[2] F. Allen. Control flow analysis. *Sigplan Notices*, 5(7):1–19, 1970.

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

[4] M. E. Benitez. Register transfer standard. TR RM-91-01, University of Virginia, March 1991.

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

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

[7] Robert P. Colwell, Robert P. Nix, John J. O'Donnel, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):318–328, August 1988.

[8] J. W. Davidson and D. B. Whalley. Ease: An environment for architecture study and experimentation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 259–260, May 1990.

[9] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler with C.* 1991.

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

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

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[13] J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for trimedia. *Journal of Instruction-Level Parallelsim*, 1(1-2), 1999. www.jilp.org.

[14] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Programming Languages and Systems*, 19(6):1031–1052, November 1997.

[15] J. Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–233, June 1992.

[16] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, July 1997.

[17] F. Mueller. Avoiding unconditional jumps by code replication. Master's thesis, Dept. of CS, Florida State University, April 1991.

[18] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 322–330, June 1992.

[19] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–66, June 1995.

[20] K. Pettis and R. C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[21] G. Ramalingam. Identifying loops in almost linear time. *ACM Trans. Programming Languages and Systems*, 21(2):175–188, March 1999.

[22] G. Ramalingam. On loop, dominators, and dominance frontier. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 233–241, June 2000.

[23] V. Sreedhar, G. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Trans. Programming Languages and Systems*, 18(6):649–658, November 1996.

[24] V. Sreedhar, G. Gao, and Y. Lee. A new framework for exhaustive and incremental data flow analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–290, May 1996.

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

[26] D. L. Weaver and T. Germond. *The SPARC Architecture Manual – Version 9.* Prentice Hall, 1994.