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

Sebastian Unger[1] and Frank Mueller[2]

[1] DResearch Digital Media Systems, Otto-Schmirgal-Str. 3, 10319 Berlin (Germany)
[2] North Carolina State University, CS Dept., Box 8206, Raleigh, NC 27695-7534
mueller@cs.ncsu.edu, fax: +1.919.515.7925

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 derived. 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: traditional and optimized node splitting as well as loop analysis through DJ graphs. Measurements show improvements of 1-40% for these methods of handling irreducible loop over the unoptimized case.

1 Introduction

Compilers heavily rely on recognizing loops for optimizations. Most loop optimizations have only been formulated for natural loops with a single entry point (header), the sink of the backedge(s) of such a loop. Multiple entry loops cause *irreducible regions* of control flow, typically *not* recognized as loops by traditional algorithms. These regions may result from goto statements or from optimizations that modify the control flow. 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, such as very long instruction word (VLIW) architectures (Phillips TriMedia, IA-64), require aggressive instruction scheduling to exploit their performance [6] but this requires knowledge about the structure of a program, which contemporary compilers generally do not support for irreducible regions of code. In addition, aggressive global instruction scheduling, enhanced modulo scheduling [13], trace scheduling and profile-guided code positioning combined with code replication [8, 9] or applied during binary translation may result in branch reordering and code replication, which itself may introduce irreducible regions. This paper briefly discusses traditional loop splitting, contributes a new approach of optimized node splitting and reports on a performance study of these approaches with DJ-graphs that recognize irreducible loops.

* Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

2 Traditional Node Splitting

Node splitting is based on T1/T2-interval analysis that detects irreducible regions in a flow graph. T1/T2 are iteratively applied on the flow graph reducing it to a simpler one:

T1 Remove any edge that connects a node to itself.

T2 A node with only one predecessor are merged into a single *abstract* node while preserving incoming edges of the predecessor and outgoing edges of the original node.

If these transformations are applied as long as possible the resulting graph is called the *limit graph*. If the final graph is trivial (a singleton node), then the original flow graph was reducible. Otherwise, all of its nodes either have none or *more* than one predecessor. Node splitting defines a transformation T3, which is applied on the limit graph:

T3 Duplicate a node with multiple predecessors (one copy per predecessor). Connect each predecessor to a distinct copy and duplicate outgoing edges of the original node.

After the application of T3, apply T1/T2 again and repeat this process. 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. 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. This work builds on Janssen and Corporaal [7] 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:

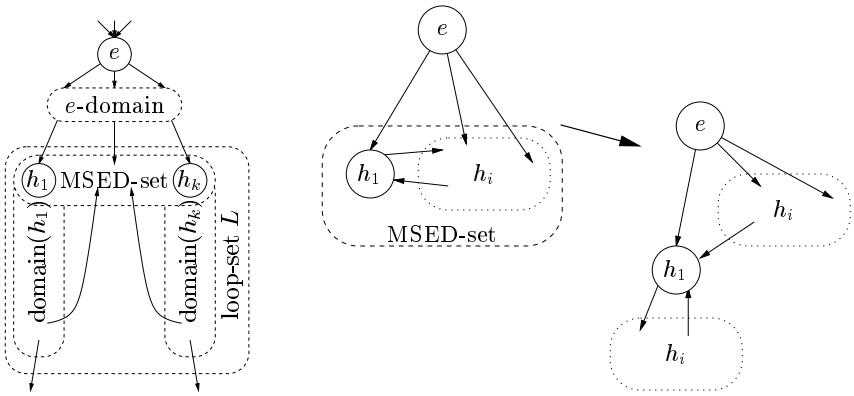
Definition 1 (Loop-set). *A loop in a flow graph is a path* $(n_1, \ldots, n_k)$ *where* n_1 *is an immediate successor of* n_k*. The nodes* n_i *do not have to be unique. The set of nodes contained in the loop is called a loop-set.*

Definition 2 (SED-set). *A Shared External Dominator set (SED-set) is a subset of a loop-set* L *whose elements share the same immediate dominator and whose immediate dominator (idom) is not part of* L*. A SED-set of* L *is defined as:* $\textit{SED-set}(L) = \left\{ n_i \in L \;\middle|\; \text{idom}(n_i) = e \notin L \right\}$.

Definition 3 (MSED-set). *A Maximal Shared External Dominator set (MSED-set)* K *of a loop-set* L *is defined as:*

SED-set K is maximal $\iff \nexists$ SED-set M, such that $K \subset M$ and $K, M \subseteq L$.

The MSED-set is a generalization of the single entry block in natural loops,



(a) Irred Loop Struct (b) One Step of the Recursive Algorithm T_r

Fig. 1. Analyzing Irreducible Loop Structures and Optimized Node Splitting

which 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 1(a) illustrates this generalized structure. Domains represent the body of a natural loop. 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').*

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:*

$$\text{domain}(h_i) = \left\{ n_j \in L \;\middle|\; h_i \text{ dominates } n_j \right\}$$

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.*

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 [12]. 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.

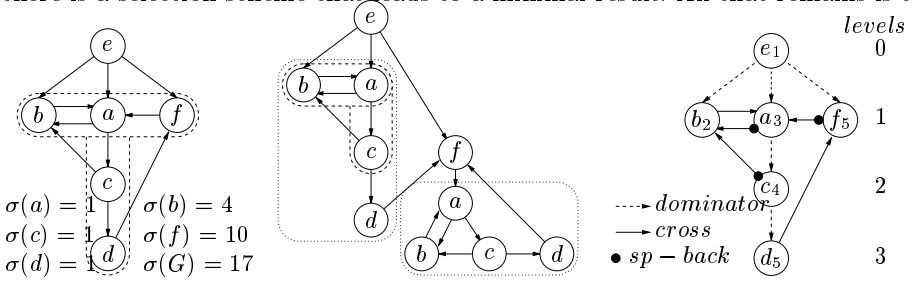
As the domains are collapsed into one abstract node, multiple edges from one domain to a single header node will reduce to just one edge from the abstract node to the header node. This reduces the number of copies of that node and is also true for multiple edges from the outside. Figure 1(a) suggests 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? Unfortunately not, as counter-examples show 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 2 gives a flow graph where no order will split the nodes to yield a minimal graph.

A new approach has been developed, based on the observation that all of the counter-examples contained one of the header nodes that was not split at all. The new approach chooses a single header node (plus its domain, of course) that *not split at all*. All other nodes of the loop-set are split once. This is illustrated in Figure 1(b). 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



(a) Orig. CFG (b) freeze f and apply T_r (c) DJ-Graph for 2(a)

Fig. 2. T3 Cannot Split this Graph with Weights σ in a Minimal Way.

actually find this scheme. First, however, 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 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:*
  - $(x, y) \in E \wedge (x, y) \notin (\text{domain}(h) \times S) \iff ((x, 1), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (\text{domain}(h) \times S) \iff ((x, 1), (y, 2)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times (N \setminus S)) \iff ((x, 2), (y, 1)) \in E'$
  - $(x, y) \in E \wedge (x, y) \in (S \times S) \iff ((x, 2), (y, 2)) \in E'$
- $s' = (s, 1)$

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. 2(a) without copying node f results in Fig. 2(b). A proof of the correctness of optimized node splitting using T_r is beyond the scope of this paper but can be found elsewhere [12]. The recursive algorithm for T_r is shown in the appendix.

5 Using DJ Graphs to Optimize Irreducible Loops

The representation of DJ Graphs [11] 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 2(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 component comprises all nodes at level 1 or below, *i.e.*, exactly one irreducible loop is found. However, Figure 2(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 2(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.

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 in loop l:

1. $dst = src$ is invariant if src is constant or its reaching definitions are outside l, as indicated by registers live on entry for *each* preheader of l.
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 l is marked invariant or if its reaching definitions are outside l.

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 dominator information of the reducible pseudo-regions to be associated with an irreducible loop.

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 are performed as for natural loops, except that invariant operations of register loads are moved into *all* preheaders of the irreducible loop.

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 [3] as a platform to conduct a performance evaluation. VPO only recognizes natural loops with a single header, just as all contemporary optimizing compilers we know of. Irreducible regions of code remain unoptimized. First, we added the recognition of DJ Graphs to VPO, extended code motion, strength reduction, induction variable elimination and recurrences. Second, we implemented optimized node splitting through T_r. 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 T1/T2/T3 was integrated. The heuristic considers for each header the number of instructions times predecessors. The header with the smallest heuristic value is then chosen.

Test programs with irreducible loops were used to measure the effect of the three different approaches. Dfa simulates a deterministic finite automata representing an irreducible loop containing two independent natural loops (see Fig. 2a in [11]). Arraymerge, extracted and translated from a Fortan application, merges two sorted arrays. The remaining programs are common UNIX utilities.

The measurements were collected for the Sun SPARC architecture using the environment for architectural study and experimentation (EASE) that is integrated into VPO. The left part of Table 1 depicts the number of dynamically executed instructions of a function that was originally irreducible. Changes in percent are reported relative to not optimizing irreducible loops. The table shows comparable reductions of 1-40% in the number of executed instructions for DJ, T3 and /TR. The improvements for $T_r$can be attributes in part to VPO, which still applies loop optimizations to reducible loops contained in the same function as irreducible ones. Other compilers may suppress optimizations resulting in even higher gains for DJ or T_r. The quantity of improvements are subject to the execution frequency of irreducible regions within the enclosing function. For

	DJ	Node Splitting		DJ	Node Splitting	
Program	Graph	T3(trad.)	T_r(opt.)	Graph	T3(trad.)	T_r(opt.)
dfa	-13.91%	-13.85%	-13.88%	-4.79%	+30.54%	+21.56%
arraymerge	-36.76%	-39.70%	-39.70%	-0.83%	+32.50%	+19.17%
tail	0.00%	0.00%	0.00%	+1.70%	+6.60%	+3.83%
unifdef1	-0.36%	+9.01%	+10.37%	+7.14%	+26.79%	+28.57%
unifdef2	-4.40%	-7.23%	-1.10%	+1.79%	+21.43%	+25.00%
hyphen	+0.01%	+0.01%	+0.01%	+8.93%	+20.24%	+20.24%
cpp	+0.05%	-1.61%	-1.18%	-0.27%	+1.64%	+2.33%
nroff1	-8.28%	-7.19%	-13.50%	+0.84%	+35.15%	+10.46%
nroff2	-4.37%	-4.42%	-0.19%	+0.47%	+25.58%	+1.86%
nroff3	0.00%	+0.06%	+0.13%	0.00%	+16.67%	+9.80%
sed	-0.59%	-4.02%	-4.49%	+2.06%	-0.40%	-1.27%
	Dynamically Executed Instr.			Static Code Size (Instr.)		

Table 1. Instructions and their Changes for Irreducible Regions

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 T_r, which is correlated to fewer delay slots of branches being filled. Similar effects were observed for cases where little changes were observed.

The right part of Table 1 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 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 T_r, 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. T3 resulted in more code growth (up to 35%). T3 mostly shows a differnt dynamic instruction count than $T_r$indicating that $T_r$reduces the amount of code duplication while preserving the performance.

The differences between the two node splitting techniques are further illustrated in Table 2 depicting the number of copied register transfer lists (RTLs) for T3 and T_r (changes relative to T3 parenthesis). Since 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 T_r, which only requires 1-30% of copied RTLs under T_r relative to T3. T3 may yield considerably inferior results that T_r for optimizing compilers with less aggressive optimizations than VPO. These findings indicate that $T_r$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.

Program(Function)	T3	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 2. RTLs copied during Node Splitting

In addition, we compared the node splitting methods T3 and T_r with the controlled node splitting (CNS) with heuristic by Janssen and Corporaal [7]. The CNS approach is detailed in the related work section. The measurements indicated that CNS differed only insignificantly from our T3 ap-

proach, 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 [8], where a miss accounts for 10 cycles delay (for going to the next memory level) 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 [4]. DJ Graphs are due to Sreedhar *et. al.* [11]. Havlak proposed a method for recognizing reducible and irreducible loops as well as the nesting of either ones [5]. 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 [10] 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 [10] 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 [10].

The notion of MSED-sets is introduced by Janssen and Corporaal [7], 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. They report reductions in code sizes to almost 1/10 of the original size. Our

measurements indicated that these savings were mostly due to the heuristic for node selection. We can also show that their formulation seldom leads to savings in practise while ours handles nested irregular regions more elegantly [12].

8 Conclusion

We derived 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. Measurements show 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.

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. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conf. on Programming Language Design and Impl.*, pages 329–338, June 1988.
4. J. Cocke and J. Miller. Some analysis techniques for optimizing computer programs. In *2nd Hawaii Conference on System Sciences*, pages 143–146, 1969.
5. P. Havlak. Nesting if reducible and irreducible loops. *ACM Trans. Programming Languages and Systems*, 19(4):557–567, July 1997.
6. J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for trimedia. *Journal of Instruction-Level Parallelsim*, 1(1-2), 1999. www.jilp.org.
7. J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Programming Languages and Systems*, 19(6):1031–1052, November 1997.
8. F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *ACM SIGPLAN Conf. on Programming Language Design and Impl.*, pages 322–330, June 1992.
9. F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conf. on Programming Language Design and Impl.*, pages 56–66, June 1995.
10. G. Ramalingam. On loop, dominators, and dominance frontier. In *ACM SIGPLAN Conf. on Programming Language Design and Impl.*, pages 233–241, June 2000.
11. V. Sreedhar, G. Gao, and Y. Lee. Identifying loops using DJ graphs. *ACM Trans. Programming Languages and Systems*, 18(6):649–658, November 1996.
12. S. Unger and F. Mueller. Handling irreducible loops: Optimized node splitting vs. dj-graphs. TR 146, Inst. f. Informatik, Humbolt University Berlin, January 2001. www.informatik.hu-berlin.de/~mueller.
13. Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, 1992.

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 changed. 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 nodes (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 [12] for more details including an adapted algorithm for finding strongly connected components (SCCs).