Maintaining Canonical Form After Edge Deletion

Eric Fritz
University of Wisconsin-Milwaukee
fritz@uwm.edu

Abstract

Waddle is a research intermediate-form optimizer that strictly maintains a canonical form similar to the loop-simplify form used in LLVM. The properties of this canonical form simplify movement of instructions to the edges of loops and often localize the effect on variables to the loop in which they are defined. The guarantee of canonical form preservation allows program transformations to rely on the presence of certain program properties without a necessary sanity-check or recalculation pre-step and does not impose an order of transformations in which reconstruction passes must be inserted.

In this paper, we present a form-preserving edge deletion operation, in which a provably unreachable branch between two basic blocks is removed from the control flow graph. Additionally, we show a distinct application of the block ejection operation, a core procedure used for loop body reconstruction, as utilized in a function inlining transformation.

1. Introduction

Waddle [8] is a proof-of-concept methodology for compiler construction and reference implementation in Scala that strictly maintains a canonical form in its intermediate representation. The canonical form is similar to the result of LLVM's loop-simplify pass. The properties of this form simplifies program transformations by allowing them to rely on the presence of certain loop properties. These properties often help contain the effects of a transformation to the loop where the transformation occurs. Every transformation over the IR is expected to maintain the properties of canonical form in addition to keeping auxiliary data structures holding domination information and loop metadata up-to-date.

This architecture differs from optimizers where transformations must canonicalize the program themselves before doing their real work as well as optimizers where transformations must declare their expectation of such canonical properties and expect the caller to ensure these properties exist before the transformation occurs. LLVM, for example, takes the second alternative approach and often requires re-calculating properties or auxiliary structures from scratch. This particular approach has the benefit of keeping the code of the transformation relevant to the transformation itself. As its contract ensures that its input has the required properties for the transformation, the transformation does not require a pre-step to canonicalize its input. Assuming that all transformations are written in the same manner, no transformation is required to maintain any properties of the input for a subsequent transformation.

However, this approach presents an obvious inefficiency. The default pass pipeline in LLVM 6.0.0 contains 240 total transformations for level $O2$ and 256 transformation for level $O3$ (some of which occur multiple times in the pipeline). Of these passes, 15 of them reconstruct the dominator tree, 12 of them re-identify natural loops and determine nesting depth, 7 of them reconstruct LCSSA form, and 8 of them canonicalize loops. Canonicalization **must** occur before transformations such as induction variable canonicalization, loop-invariant code motion, loop rotation, loop sinking, and loop unrolling. The number of passes to maintain canonical form and update required data structures account for nearly one-sixth of the number of passes in this pipeline.

Some passes attempt to 'repair' structures such as the dominator tree modified by the transformation when the changes to such structures are sufficiently localized in order to reduce the number of from-scratch calculations required down the line. Many of these repairs are ad-hoc. A fair amount of research [1, 3, 6, 9, 12–15, 17–19] spanning over two decades has gone into incrementally maintaining the dominator tree of a control flow graph as flow paths are modified. Waddle expands this idea of incremental maintenance to the loop nesting forest, LCSSA form, and canonical form.

Intuitively, one would expect transformations in Waddle to be littered with maintenance logic – but this is not the case. Waddle uses the observation that not all modifications of the control flow graph make sense when the goal is to preserve semantic equivalence. For example, arbitrary edge additions between two existing blocks are exceedingly rare. Such arbitrary modifications can also complicate the identification of loop structures [10, 20, 21]. Operations that modify the control flow graph can then be categorized into two groups: additive and subtractive. Every operation additionally preserves canonical form of the control flow graph, thus their composition with other operations and their inclusion in more complex transformations provides this preservation benefit 'for free'. We briefly describe each group to give a more substantial view of Waddle.

Additive operations are rather restricted and involve cloning an existing single-entry portion of the program and linking it back into the graph. These operations are useful for optimizations such as function inlining which involves duplicating an entire function, and loop switching and unrolling which involves duplicating a loop body. When creating a duplicate subprogram, the domination and loop structures associated with the original blocks are cloned as well and inserted into the target function. Any branch to a block that is not marked for cloning simply points to the original block. Generally, linking this subprogram into the graph leaves only local inconsistencies in domination and loop nesting structures where the single-entry clone hooks into the original graph, and at each 'exit' block of the duplicate region. Thus, only a small handful of things

Submitted to ICOOOLPS, 2018.

could be inconsistent with canonical form during these restricted additive operations.

The subtractive operations simply delete a block or an edge from the control flow graph or a function from the program. These operations are most useful for the elimination of dead or inaccessible code and often present additional opportunities for optimization. This paper presents the case of deleting a single (presumably untaken) edge from the control flow graph, which falls squarely into the latter category.

The remainder of this paper is organized as follows. Section 2 gives an overview of loop nesting forest structure and defines the canonical form of a control flow graph. Section 3 discusses the edge deletion operation. Section 4 discusses additional uses of the edge deletion operation in transformations such as function inlining. Section 5 gives a brief evaluation of this technique comparing incremental reconstruction and from-scratch re-computation of the loop nesting forest over several programs. Section 6 discusses similar work in the context of incrementally maintaining dominator trees and Section 7 discusses relevant implementation details.

2. Loop Nesting Forests and Canonical Form

In this section, we define canonical form and provide incentive for its use. First, we define a handful of prerequisite terms.

Natural Loops and the Loop Nesting Forest Consider the following program in which two sequences X and Y are searched for a common element. We assume that both sequences are non-decreasing so that iteration of the outer loop can be avoided when the maximum element of X is less than the minimum element of Y, and iterations of the inner loop can be avoided when $x < y$. Program points referenced later are labeled in the right gutter.

```
if |X| ≠ 0 ∧ |Y| ≠ 0 ∧ X_{|X|−1} ≥ Y_0 then                ▷ s
    outer:
    for x ∈ X do                                          ▷ a
        for y ∈ Y do                                      ▷ b
            if x = y then                                 ▷ c
                return true                               ▷ d
            else if x < y then                            ▷ e
                continue outer
            end if
        end for                                           ▷ f
    end for
end if
return false                                              ▷ g
```

A (simplified) control flow graph for this program is illustrated in Figure 1. This graph contains two natural loops defined by header blocks a and b. Concisely, a *natural loop* is a maximal strongly connected component in the graph where each block is dominated by a single block called the *header*. Natural loops have a single point of entry. Loops with multiple entry points may occur if the control flow graph is irreducible [7, 10, 11, 21, 23], but we focus our attention on the reducible case. The blocks which compose the body of each loop are outlined.

Each loop has an additional set of properties derived from the flow graph. The *body* is the maximal strongly connected set of blocks dominated by the header. A *latch* is a predecessor of the header contained in the loop body. A *backedge* is a directed edge from a latch to the header. An *exit* is a block not contained in the loop body with a predecessor contained in the loop body. A loop may also have a *preheader*, a dominating predecessor of the header from which all flow paths enter the loop.

If the body of a loop l is a proper subset of the body of loop l', then l is said to be *nested* in l', and l is a child of l' in the *loop nesting forest* of the function. From this definition, it is clear that the loop nesting forest forms a subset lattice.

In our example control flow graph, the *outer* loop consists of blocks a, b, c, e, and f. It has two latches (e and f) and two exit blocks (d and g). Block s is not a preheader of this loop as there is a path from s which can avoid the loop. The *inner* loop consists of blocks (b, c, and e). It has one latch (e) and three exits (a, d, and f). Block a is a preheader of this loop.

The body set, exit set, and nesting relationship of a loop can be easily re-computed on demand from the control flow graph. However, it is very convenient to maintain these sets for efficient containment and subset queries. In the following, when we *repair* the body set of a loop, we are modifying the cached set to match the definition.

LCSSA Form Loop-closed static single assignment form is an extension of SSA form in which all uses of a register occur within the loop where that register is defined. This has the effect that transformations on a loop may remain local – transformations do not need to exit the loop to otherwise modify uses of a register that was somehow affected. Along with maintaining the dominator tree, loop nesting forest, and canonical form, we also take care to ensure graphs conform to LCSSA form after a transformation.

Canonical Form We say a reducible control flow graph is in canonical form if every natural loop l in the graph conforms to the following properties.

Property 2.1. Loop l contains only one latch.

Property 2.2. Loop l has a dedicated preheader.

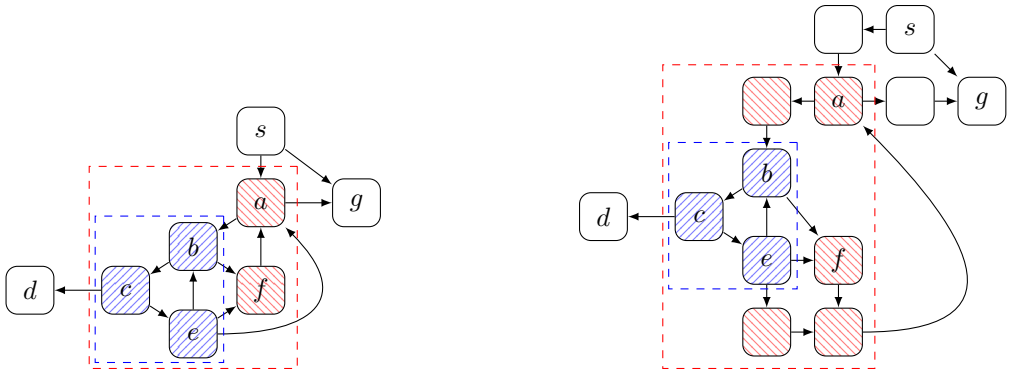
Property 2.3. Every exit of loop l is dedicated to l.

In this context, *dedication* of a block b to a loop l requires that if b is a predecessor of a block in l, then all successors of b are in the body of l and, symmetrically, if b is a successor of a block in l, then all predecessors of b must be in the body of l.

A graph can easily be *canonicalized* by a process called *edge set splitting* [8]. Simply, a new no-instruction block is inserted on a set of edges that share a common target block. We give concrete examples of this process in the following. The result of canonicalization of our example control flow graph is also given in Figure 1.

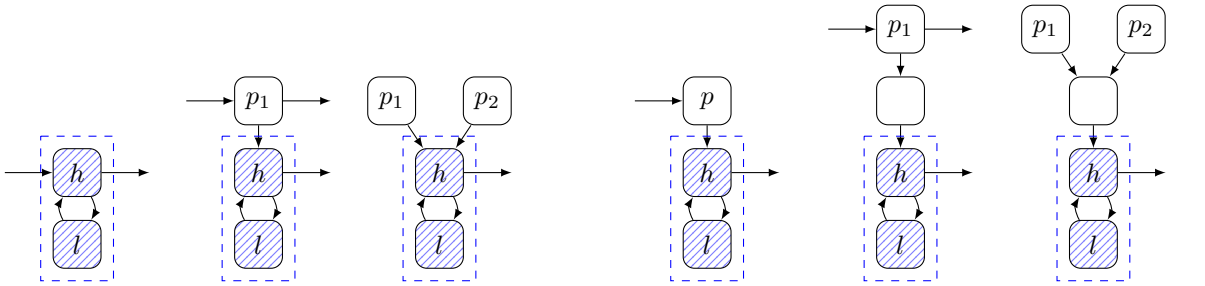
Property 2.1 ensures that the loop can be uniquely defined by its backedge. This can simplify loop iteration/trip count estimation, as every subsequent pass through the loop must travel through the same edge. Combined with the next property that guarantees a preheader, the loop header is guaranteed to have exactly two predecessors. This simplifies the recalculation of loop body membership after a change to the paths in the control flow graph (for continued membership queries, it only needs to be shown that there is still a path from the block to a latch). Our example program violates this property as both block e and block f are latches of the outer loop. This property can be repaired by replacing the reference to block a from both blocks with a reference to an additional empty block which then jumps unconditionally to block a. This fresh block becomes the unique latch of the outer loop.

Property 2.2 simplifies the correctness of *hoisting* a loop-invariant instruction out of a loop body. To maintain (order-semantic) similarity, an instruction hoisted from a loop must be moved into every predecessor of the loop header, otherwise there would exist a path in the control flow graph in which the loop is entered and the instruction not evaluated. If there exists a path that contains such a predecessor but avoids the loop, then evaluation of the instruction now occurs on a path that does not use the value. This can negatively affect performance of the program if the instruction is expensive or the loop is infrequently entered



(a) Unmodified control flow graph. (b) Canonicalized control flow graph.

Figure 1: Control flow graph of the sample program.



(a) Sample concrete violations. (b) Violations repaired by insertion of additional empty blocks.

Figure 2: Violations that prevent efficient instruction hoisting.

(i.e. the altered path is hot). If there are multiple such predecessors, then the hoisted instruction is duplicated, increasing program size. In general, duplication of instructions cause a second assignment of a register to be introduced, violating static single assignment form (which, if expected by future transformations, must be reconstructed). Control flow graphs violating this property are illustrated in Figure 2. Our example program violates this property twice: neither s nor a are dedicated preheaders. In order to dedicate the preheader of the outer loop, an additional empty block must be inserted on the edge (s, a). Similarly, the preheader of the inner loop can be dedicated by adding an additional empty block on edge (a, b). These fresh blocks become the dedicated preheader of the outer loop and inner loops, respectively.

Property 2.3 makes it possible to safely *sink* an instruction from a loop body to the loop exits (e.g. saving a write to a memory address unread within the loop until the loop termination). The same instruction duplication issues present with instruction hoisting are also present with instruction and effect sinking. Even more concerning, an instruction sunk to an exit not dominated by the loop header may be malformed as there may be missing data dependencies that are present only on paths through the loop. Control flow graphs violating this property are illustrated in Figure 3. It is important to note that an instruction sunk to multiple exits, even when in canonical form, may need to be rewritten to preserve SSA form. In our example program, block f is a dedicated exit of the inner loop as all of its predecessors are in the body set of the inner loop. Similarly, block d is a dedicated exit of both the inner and outer loops. Block g, however, is not a dedicated exit of the outer loop due to the edge (s, g). In order to dedicate the exit, an additional empty block must be inserted on the edge (a, g). This fresh block becomes a dedicated exit of the outer loop, and g is no longer an exit of the outer loop. Later, we refer to this process of dedicating an exit block e to a loop l by adding additional blocks in pseudocode as $\mathit{dedicate\ exit}(l, e)$.

When repairing a single violation of a canonical form property, the number of total violations of the graph decreases by one. In the example, creating a unique latch by splitting the edge set $\{(e, a), (f, a)\}$ creates a new block that is an undedicated exit of the inner loop. However, block a is no longer an exit of the inner loop. In this particular instance, the block which violates a property had changed, but the number of total violations did not increase.

Each of these properties can be constructed fairly easily when needed given a reducible control flow graph and an accurate loop nesting forest. We maintain these properties at all times in order to simplify the implementation and proofs of preserved semantics of Waddle as well as the implementation of other transformations that rely on canonical form properties. While preservation of these properties is useful in its own right, these properties also simplify the preservation of the loop nesting forest.

3. Edge Deletion

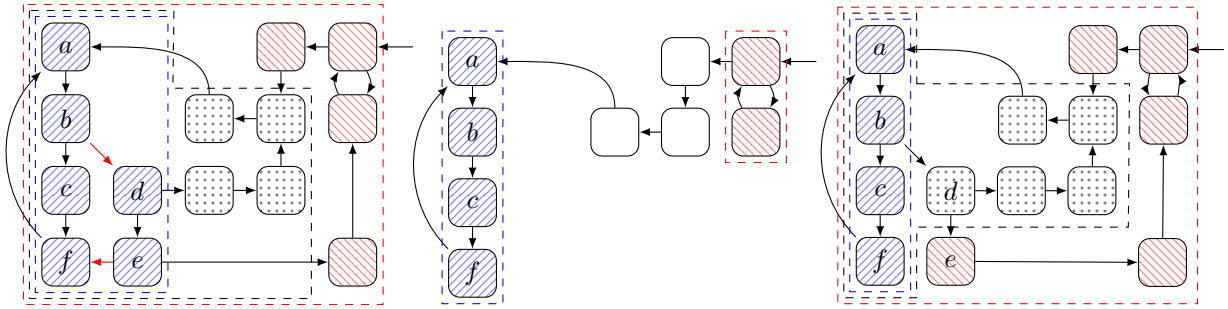
In this section, we describe the algorithm which repairs the loop nesting structure and canonical form after the deletion of an edge (a, b). We also describe *block ejection*, a subprocedure applied to a loop l to do some of the heavy lifting of edge deletion. The later procedure has additional applications outside of edge deletion, one of which is discussed in Section 4.

3.1 Edge Deletion

A program may have a control flow edge which is provably *dead* such that no set of inputs will cause the evaluation of the program

(a) Sample concrete violations. (b) Violations repaired by insertion of additional empty blocks.

Figure 3: Violations that prevent efficient instruction sinking.



(a) The original graph. (b) Graph after removal of edge (b, d). (c) Graph after removal of edge (e, f).

Figure 4: Examples of loop structure changes that can occur after an edge deletion.

to take that edge. Such edges often occur as a result of another optimization pass such as function inlining or loop unswitching. Removing this edge results in a reduction of program size, and may allow additional opportunities for further optimization (reducing the size of a function below the inline limit, removing dead side effects from a loop for more precise analysis, etc). This section details the restoration of canonical form after the deletion of such a dead edge.

The deletion of an edge has what appears to be non-local changes to the graph. Many nodes can become unreachable if all paths to them contain the deleted edge. Entire loops may be destroyed or un-nested from their parent. Figure 4 illustrates such changes. Removing the edge (b, d) from the initial graph causes the middle loop to lose its backedge, causing the set of blocks composing the loop body to become disconnected. The innermost and outermost loops survive, but the inner loop is no longer nested within the outer loop as there is no path from a block in the inner loop's body to the outer loop's latch. In this case, all paths were destroyed, but in general unnesting occurs if there is no path to the outer loop's latch that does not exit the outer loop's body. Removing the edge (e, f) from the initial graph forces blocks d and e to be ejected from the inner loop into ancestor loops.

The deletion algorithm, shown below, can be organized into three steps described in the following.

Step 1 (lines 1-4) A reference to block b from a terminating instruction of block a is removed. We do not cover the case where the sole edge exiting a is deleted – in order to keep the control flow graph complete, such a branch instruction must be replaced by a return instruction. Thus, we assume that the branch instruction of a references several targets (one or more of which is block b). If block a has multiple edges to block b, then the deletion of one such reference from a does not meaningfully alter any path of the control flow graph. Thus, no reconstruction of the dominator tree, loop nesting forest, LCSSA form, or canonical form is necessary.

Algorithm 1 Edge Deletion

```
1:  Remove a reference to b from a
2:  if edge an (a, b) still exists then
3:      return
4:  end if

5:  R ← repair dominators(a, b)
6:  K ← {l | latch(l) ∈ R or (a, b) is the backedge of l}
7:  for l ∈ K do
8:      Remove l from forest
9:  end for
10: for b′ ∈ R do
11:     Remove b′ from all loop body sets
12:     Remove b′ from all loop exit sets
13: end for
14: for l ∉ K where parent(l) ∈ K do
15:     p ← lowest ancestor of l such that p ∉ K
16:     Attach l as a child of p
17: end for

18: p ← smallest loop containing a such that p ∉ K
19: eject(p)
```

Then, the algorithm is complete. The remaining steps assume this is not the case.

Step 2 (lines 5-17) First, the dominator tree is repaired using the algorithm described by Ramalingam and Reps [17] or a similar algorithm. This particular algorithm is chosen as it yields the set of newly unreachable blocks, denoted by R. However, another algorithm could be chosen if the set R is calculated first – in brief, R is the empty set if b is still reachable after the deletion of edge

(a, b), and is the set of blocks in the (old) dominator subtree rooted at b otherwise.

The loop body and loop exit sets are pruned to remove references to unreachable blocks and the loop nesting structure is pruned to remove references to loops that have lost their backedge. At the end of line 17, the current loop nesting forest forms a fairly good approximation of the correct loop nesting forest which can later be refined. This approximation has the following five properties.

1. The loops of the function and loops in the approximate loop nesting forest have a one-to-one correspondence, although the loop may have an inaccurate parent, body set, or exit set.
2. The body sets of the approximate loop nesting forest maintain a subset lattice (just as well-formed loop nesting forests do).
3. The body set of each inaccurate loop in the approximate loop nesting forest is a superset of the body set of the loop in the function with the same header.
4. The exit set of each inaccurate loop in the approximate loop nesting forest is correct with respect to body set of the same loop – specifically, the exit set must consist of all the blocks that have a predecessor in the approximate body set but are not themselves in the approximate body set.
5. Every loop with an inaccurate body set is an ancestor of the deepest inaccurate loop in the approximate loop nesting forest. More specifically, all inaccurate loops must have contained block a prior to the deletion of the edge.

Before refining the approximation, we give a brief proof that each of these properties hold in the approximate loop nesting forest.

1. Let l be a loop of the function prior to the deletion of edge (a, b) and let S describe the subgraph of the loop reachable from $header(l)$ after the deletion of edge (a, b). S remains strongly connected if it contains the block $latch(l)$ and (a, b) was not the backedge of l. Notice that $latch(l)$ is reachable only if it remains reachable from $header(l)$ due to domination. This is the same as the condition $l \notin K$. Additionally, the deletion of an edge cannot introduce a backedge to the flow graph, so no loop can be introduced to the function.
2. The body sets of the approximate loop nesting forest are constructed by removing blocks from the loop nesting forest prior to the deletion of edge (a, b). When a block is removed from a loop, it is also removed from all of its descendants. As the original loop nesting forest was a subset lattice by definition, the approximate loop nesting forest also forms a subset lattice.
3. Let l be a loop in the approximate loop nesting forest whose body set is equivalent to $body(l) \setminus R$ by construction. Suppose a block $b' \notin body(l) \setminus R$ belongs in the body set of l after the deletion of the edge. There must then exist a pair of paths $header(l) \to b'$ and $b' \to latch(l)$, the two of which which did not exist. This cannot occur as edge deletion does not add paths to the flow graph.
4. Let L and X be the approximate body and exit sets, respectively, of a loop l. First, suppose $pred(b') \cap L = \emptyset$, but $b' \in X$. As X is constructed only by removing unreachable exits, b' was necessarily an exit of loop l prior to the deletion of the edge and as the function was in canonical form, this was a dedicated exit. As there are no remaining edges from L to b', $b' \in R$ and b' cannot exist in X. Now, suppose $pred(b') \cap L \neq \emptyset$ and $b' \notin L$, but $b' \notin X$. The absence of b' in X implies that b' was not an exit of l prior to the deletion of the edge and no such exit path can be constructed by the deletion of an edge.

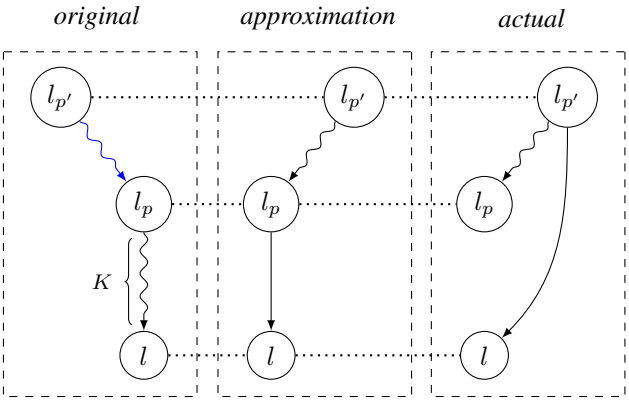


Figure 5: The relation between approximate and true nesting structures after the deletion of an edge. The loops corresponding between forests are shown by dotted lines.

5. Let l be a loop in the claimed loop nesting forest and let b' be a superfluous block in the claimed body set of l that was correctly in the body set of l before the deletion of edge (a, b). Now, suppose a is not in the body set of l. Then, the deletion of edge (a, b) either changed no paths in l, or made $header(l)$ unreachable. Both of these cases present a contradiction: in the former case, l's body set does not contain any superfluous blocks; in the later case, there is no entry in the claimed loop nesting forest representing loop l.

These properties lead us directly to an algorithm for refining the approximation, presented in detail in Section 3.2. By Property 1, the approximate loop nesting forest has already identified the correct loop headers and, if the control flow graph was in canonical form prior to the deletion of the edge, the correct loop backedge. This, combined with Property 5, allows us to bound the loops that need to be refined: the loops encountered tracing from the deepest loop that had contained block a to a root of the approximate loop nesting forest. Figure 5 describes the relationship between violating loops in the claimed loop nesting forest. In this example, loop l is actually a direct child of loop $l_{p'}$ (this occurs as the blocks of l no longer have a path to the latch of loop l_p).

The remaining properties describe the manner in which a loop may be inaccurate. Property 3 ensures that the body set of a loop can be refined simply by removing nodes. Property 2 and Property 4 show, respectively, that the parent and the exit set of a loop are correct *unless* blocks are removed from the body. When blocks are removed from a loop body, it is trivial to determine the correct parent (the closest ancestor that remains a superset) and the correct exit set can be determined by looking only at previous exits and removed blocks.

Step 3 (lines 18-19) The *eject* procedure refines the approximation constructed in the previous step.

3.2 Block Ejection

Now, we turn our focus to refining the approximate loop nesting forest created in the previous section so it becomes the correct unique loop nesting forest of the modified control flow graph. We have two major concerns to address: removing extraneous blocks in the body set of an approximate loop, and placing loops under the correct parent so that the loop nesting structure forms a subset lattice.

Figure 6 illustrates a repair of such a loop nesting forest. The graph of this example may have been created by removing the edge

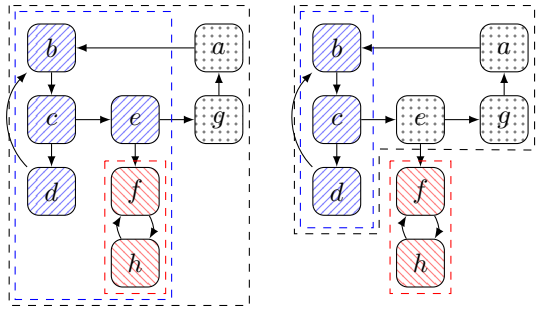


Figure 6: Repairing loop nesting forest by ejecting blocks from a loop.

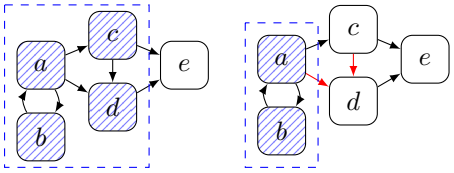


Figure 7: Block ejection creating an undedicated exit.

(f, d) or by inserting the subgraph consisting of blocks e, f, and h on the edge (c, g), among other possibilities. Section 4 discusses how function inlining can create such a broken graph. To repair the graph, block e is ejected from the middle loop (but remains in the outer loop as a path to the latch of the outer loop exists) and blocks f and h are ejected from both the middle and outer loops. This transformation also alters the exit sets so that g is no longer an exit of the middle loop but e is, and f is the sole exit of the outer loop (which had no exits previously). Notice that the entire body of the innermost loop was ejected from its parent loop, and therefore the parent of the ejected loop structure also changes.

The ejection algorithm, shown below, recurses over loops, starting from the deepest inaccurate loop and working its way up to a root of the forest. For each loop l, we partition its body set into the disjoint sets I and O. Set I is composed of the blocks from which the loop's latch is still reachable in the subgraph induced by $body(l)$ (equivalently, where the loop's latch is still reachable via a path that does not contain $header(l)$). Set O is composed of the blocks that are no longer part of the connected component forming l after the deletion of the edge.

The extraneous blocks are pruned from the body of the loop. This only affects the current loop – all blocks pruned from l are still members of the body of all of l's living ancestors. They may, however, be ejected from an ancestor on a subsequent recursive call. If the header of a loop is ejected (and then, necessarily, the entire loop body), the ejected child loop is un-nested from l and re-attached to l's closest living ancestor.

Then, the exit set of loop l is recalculated to determine which newly ejected blocks are new exits and which old exits no longer have predecessors in the loop body.

It may also be helpful to note that if ejection is called as part of an edge deletion procedure, the unreachable blocks have already been pruned from the function. In this context, all blocks in the claimed body of a loop are reachable from the header of that loop due to domination.

Block ejection may create a violation of the dedicated exit property. Figure 7 illustrates this case in which block d and one of its predecessors are ejected from the loop. The *dedicate exit* operation is applied over violating exit blocks in order to repair the

Algorithm 2 Block Ejection

$I \leftarrow \{b \in body(l) \mid$ there exists
    a path $b \rightarrow latch(l)$ in subgraph induced by $body(l)\}$
$O \leftarrow body(l) \setminus I$

for $b \in body(l)$ **where** $b \in O$ **do**
    Remove b from $body(l)$
end for
for child loop l' of l **where** $header(l') \in O$ **do**
    Attach l' as a child of $p(l)$
end for
for $b \in exit(l)$ **where** $pred(l) \cap I = \emptyset$ **do**
    Remove b from $exit(l)$
end for
for $b \in O$ **where** $pred(b) \cap I \neq \emptyset$ **do**
    Add b to $exit(l)$
    $dedicate\ exit(l, b)$
end for
for each definition d of r
  where d occurs in I and a use of r occurs in O **do**
    $repair\ lcssa(l, d)$
end for

if l is not a top-level loop **then**
    $eject(parent(l))$
end if

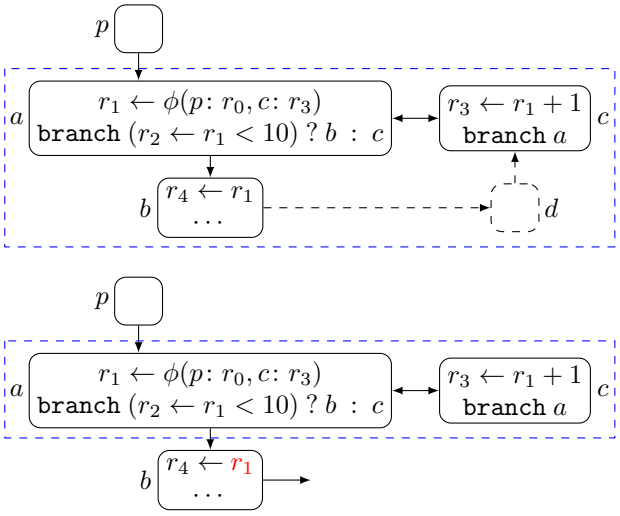
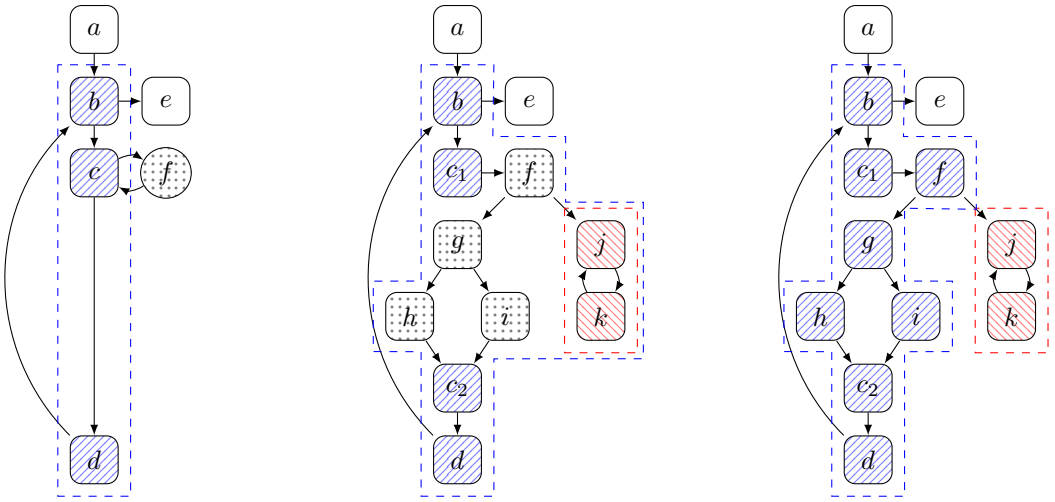


Figure 8: Block ejection creating an LCSSA violation.

temporary violation of canonical form. As a reminder, the *dedicate exit* procedure referenced in the algorithm pseudocode was briefly discussed in Section 2.

Block ejection may also create a violation of LCSSA form. Figure 8 illustrates this case in which a block containing a legal use of a register is ejected from the loop that defines that register. An LCSSA reconstruction algorithm, such as the one presented by Braun et al. [5], is applied over such violations to repair LCSSA form. The details of this operation are beyond the scope of this paper.



(a) Pre-inlining. (b) Post-inlining. (c) Post-ejection.

Figure 9: Function inlining creates an opportunity for block ejection.

4. Additional Applications

Edge deletion is a common operation in optimization, but is not the only application of the block ejection technique. Another prime opportunity for block ejection occurs during function inlining. When a function is inlined, the entire domination and loop nesting structures of the inlined function are transplanted into the same structures of the calling function. If the inlined function has blocks that cannot reach a function exit (non-exiting functions or non-exiting loops), blocks and loops will be placed too deeply in the resulting loop nesting forest. This section details how to leverage block ejection to solve this edge case.

The first two graphs of Figure 9 illustrates function inlining and shows its effects on the loop nesting forest. The block containing a call to a known function f can be split in two so the instruction that occurs immediately before the call terminates block c_1 and the instruction occurring immediately after the call begins block c_2. Block c_1 then branches to the entry block of a clone of f, and each exit block (terminating with a `return`) of the clone instead branches back to c_2.

Fortunately, the dominator tree changes only locally. A clone of the dominator tree of f must be attached as a child of the dominator node of c_1, as c_1 immediately dominates $entry(f)$ and the paths within f do not change. The new immediate dominator of c_2 is the lowest common ancestor of all exit blocks of f, due to an observation made by Alstrup and Laurisden [2]. As paths are only altered between blocks c_1 and c_2, no other domination relationships change and the dominator tree structure is maintained.

In the general case, the loop nesting forest can also be updated simply. All blocks of f are inserted into the body of the loop containing blocks c_1 and c_2, and a clone of the loop nesting forest of f must be inserted as a child of the same loop. Assuming all paths in the inlined function can reach *some* exit of the inlined function, this is a correct transformation: as all blocks of f are reachable by c_1 and c_2 is a successor of all exits of f, all blocks of f must belong in the same connected component of c_1 and c_2. However, it may be the case that the inlined function contained a block that had no path to any exit of f. This also guarantees that no such blocks can reach block c_2, nor the latch of the loop containing c_1 and c_2. Thus, the loop in which the function was inlined now contains extraneous blocks that must be ejected. This process is illustrated by the later two graphs in Figure 9. In this example, blocks e' and f' form a closed loop from which d is unreachable. These blocks must be ejected to preserve the loop nesting forest.

5. Evaluation

As a brief evaluation to show the performance benefits of this technique, we create a set of non-trivial flow graphs, then perform a sequence of edge deletions. Each edge deletion uses the output of the previous deletion, and all deleted edges are guaranteed to still exist at the time of their deletion (no non-effectful operations were counted).

For baseline performance, we calculate (from scratch) the dominator tree and loop nesting forest and test each loop for canonical form (re-canonicalizing the loops in violation) after each edge deletion. To contrast the baseline, we use the incremental technique described in Section 3 to repair the dominator tree and loop nesting forest and maintain canonical form.

The flow graphs of the Waddle IR used for this evaluation were generated from compiled LLVM IR. The resulting flow graphs are (mostly) isomorphic in control flow to the LLVM IR, but each instruction in the generated graph is chosen randomly. Exception flow, indirect function calls, and unreachability, for which there are currently no direct representations in the Waddle IR, were approximated as closely as possible. An instruction in a basic block of the source IR and the generated instruction in the symmetric basic block of the generated IR use the same register names.

The resulting generated graphs have the same control flow structure and live register intervals of a real-world program. This is more than sufficient for this benchmark, as the exact semantics of the program are inconsequential.

The six programs used to generate the LLVM IR are single-file C++11 implementations of graph algorithms taken from the GAP Benchmark Suite [4], described below. These sources were chosen because they utilize a wide variety of iterative structures that were likely to make realistic yet interesting loop nesting forests. The original source (git commit `f166dc4`) is available on Github[1]. Each compilation unit performs an operation over a graph whose edges are stored as compressed sparse rows.

[1] `https://github.com/sbeamer/gapbs`

source	recompute all	repair all	recompute dom	repair dom	% savings claimed
bc.cc	72.308 ± 18.022	19.982 ± 4.568	22.490 ± 5.980	1.720 ± 0.410	63.34%
bfs.cc	45.898 ± 1.132	14.678 ± 3.168	17.232 ± 1.498	1.750 ± 0.820	54.90%
cc.cc	59.614 ± 13.836	16.266 ± 2.074	18.664 ± 5.316	1.664 ± 1.146	64.34%
pr.cc	45.618 ± 5.062	16.058 ± 4.812	14.868 ± 3.272	1.500 ± 0.670	52.66%
sssp.cc	48.704 ± 6.026	14.650 ± 0.580	16.154 ± 2.464	1.384 ± 0.184	59.24%
tc.cc	51.884 ± 10.716	16.688 ± 3.042	17.436 ± 3.544	1.564 ± 0.316	56.10%

Figure 10: Wall time required to run benchmark – all times are given in milliseconds.

source	edges	domination queries	nca queries	dominator iterations	partition iterations
bc.ll.ir	716	1,435	2,940	1,263	3,549
bfs.ll.ir	565	1,176	2,424	1,026	2,401
cc.ll.ir	601	1,215	2,474	1,066	2,422
pr.ll.ir	583	1,195	2,390	1,029	2,253
sssp.ll.ir	581	1,175	2,443	1,038	2,146
tc.ll.ir	595	1,214	2,480	1,061	2,342

source	edges	domination queries	idom calculations	backedge queries	gather iterations
bc.ll.ir	716	76,775	44,454	22,943	17,253
bfs.ll.ir	565	43,353	35,164	18,147	8,147
cc.ll.ir	601	55,010	35,218	18,210	11,949
pr.ll.ir	583	39,236	31,356	16,261	7,847
sssp.ll.ir	581	42,644	33,098	17,130	8,758
tc.ll.ir	595	52,944	35,922	18,556	10,371

Figure 11: The number of (interesting) operations performed during benchmark trials (the top specifies the incremental repair technique, and the bottom specifies re-computation technique).

- `bc.cc`: calculate betweenness centrality scores for each vertex
- `bfs.cc`: perform a breadth-first traversal and create a mapping vertices to its predecessor in the traversal order
- `cc.cc`: label each vertex with an identifier for its connected component
- `pr.cc`: calculate PageRank scores for each vertex – terminate once the total change is less than some epsilon
- `sssp.cc`: calculate the shortest path distance from a source vertex for all vertices
- `tc.cc`: count the number of triangles (cliques of size 3) in an undirected graph where neighborhoods are sorted by vertex labels

To give a brief sense of the scale of the Waddle IR used here: each compilation unit contains between 363 and 445 total function definitions, between 71 and 101 of which are interesting (neither trivial, a single node, nor loop-less). There are between 2 and 237 blocks per function (with an average of 21 blocks per function) and between 2 and 354 edges per function (with an average of 30 edge per function). Each interesting function contains between 1 and 10 loops with depths ranging between 1 (top-level) and 4. The size of loop body sets range between 1 and 99 blocks, and the size of loop exit sets range between 0 and 10 blocks. Many of the loops with empty exit sets come from Waddle's equivalent of an *unreachable* LLVM instruction, in which a single trivial block branches back to itself.

Figure 10 gives the wall-time required for running the sequence of edge deletions over each of the programs in the suite. The *repair* columns give the time taken for the incremental approach, and the *recompute* columns give the time taken when the dominator and loop structures are computed from scratch between operations. We break this down further into the time required for the entire operation (the *all* columns), and the time required only to repair the dominator tree (the *dom* columns) to show that the time savings are not completely due to previous work in dominator tree reconstruction. The *% savings claimed* column gives the percentage of the average runtime reduction that is **not** attributed to previous work (i.e. the percentage reduction between the difference of the *recompute all* and *repair all* columns and the difference of the *recompute dom* and *repair dom* columns). Each data point is an average of five trials. For this set of programs, the incremental runtime runs in about 30% of the time required for the baseline approach on average.

To examine why the runtime of the incremental approach is so much better, we can compare the number of interesting operations used by each technique. In this context, we use interesting operations to mean those with a dominating usage count or those that are likely to be expensive to perform. Figure 11 gives these details (where non-interesting operation counts are omitted). In both tables, the *edges* column specifies how many edges were in the deletion sequence for each source program.

Both techniques performed a fair number of domination queries, but the incremental technique showed a 97% decrease on average in the total number of such operations performed. The remaining interesting operations diverge for each technique.

For the incremental repair technique, *nearest common ancestor query* and *dominator iteration* operations are both used to repair the dominator tree. In lieu of maintaining priority order of the control flow graph, we alter the Ramalingam and Reps [17] dominator tree repair algorithm to simply perform a fixed-point loop during reconstruction – this change is discussed further in Section 6. The *partition iteration* operation specifies the number of set intersections required to determine which blocks remain in the body of a loop after an edge deletion. This, in essence, counts the number of levels visited in a breadth-first search from the latch of a loop.

For the re-computation technique, the *idom calculations* are used to compute the dominator tree. The *backedge query* operation is used to find loop header candidates by checking whether or not

they dominate one of their predecessors – this explains the massive increase in domination queries for this technique. Once a header is found, its body is found via a series of *gather iteration* operations. Similar to partition, gather performs the same breadth-first search from a latch back to the header candidate. When comparing the number of partition and gather iterations between the two techniques, there is a 75% decrease on average in the number of iterations performed by the incremental technique.

We find these results to be very encouraging despite the baseline methodology being rather unsophisticated. A more intelligent baseline would batch such deletion operations so that very frequent recomputation is unnecessary.

Waddle uses the edge deletion operation in a number of orthogonal transformations (currently if-simplification, jump threading, jump simplification, function inlining, and loop unswitching). Batching deletions across different optimization passes would prove difficult as each optimization pass assumes that canonical form has been preserved in its input. Additionally, this evaluation has shown that when reconstruction or iterative repair is actually necessary, the savings can be significant – the savings of repairing the loop nesting forest alone is very similar to the savings of repairing the dominator tree. The latter operation has received decent research and implementation attention (which implies its worth in practice). More robust evaluations that include the savings over entire optimization passes, which would conform precisely to the usage of an optimizing compiler in practice, are planned for future work.

6. Related Work

Computing the loop nesting forest efficiently has been a well-researched topic [10, 16, 20–22], but research into incrementally maintaining the the loop nesting forest as the flow graph it represents undergoes changes remains wanting. The same cannot be said for incremental re-computation of dominator trees, which has received an abundance of research attention over the years [1, 3, 6, 9, 12–15, 17–19]. In this section, we briefly describe an approach presented by Ramalingam and Reps [17] to update the dominator tree of a reducible control flow graph when a single flow edge is inserted or removed. We focus on this particular algorithm due to its use in the proof-of-concept implementation of Waddle.

The basic idea of the algorithm is to conservatively approximate a set of *affected* blocks for which the immediate dominator changes, then re-calculate the new immediate dominator using the old dominator tree.

The algorithm requires only a single pass if the affected blocks are processed in a topological order of the acyclic graph induced by removing backedges. This ordering is encoded by giving each block a *priority* and updating those priorities whenever an edge is added to the graph. Removal of edges do not affect block priorities.

Let n be the number of blocks in a graph and let m be the number of edges. The worst-case complexity of updating the dominator tree (excluding the step of updating the priority ordering when necessary) is $\mathcal{O}(\|A\|^{\leftrightarrow} \log(n))$[2], where A is the approximate set of affected blocks. Alpern et al. [1] present an incremental algorithm to update the priority orderings of a graph after the insertion of an edge, which takes $\mathcal{O}(\|\kappa\|^{\leftrightarrow} \log(\|\kappa\|^{\leftrightarrow}))$ time in the worst case, where κ is the set of blocks whose priorities will change. In all, updating the dominator tree after inserting or removing a single edge takes $O(m \log(n))$ time.

The deletion of a backedge does not affect domination and requires no update to the dominator tree. The deletion of a forward edge (u, w) where w is still reachable may reduce the set of paths to a block y, which changes its immediate dominator. The affected blocks are conservatively approximated by the following set, which contains the blocks that are siblings of w in the dominator tree.

$$\{v \mid idom(v) = idom(w)\},$$

An example of a dominator tree's affected blocks is illustrated by Figure 12.

The immediate dominator of each block in this set is then re-calculated. If the deletion of edge (u, w) causes the subgraph R (which consists of the blocks dominated by w) to become disconnected, then each block is removed, and any forward edges from R to a still-reachable block must be deleted individually as described above.

7. Implementation Details

In the reference implementation, a block keeps a reference to the set of its predecessors and a reference to the deepest loop to which they belong. Finding the body sets to which a block $b \in R$ belongs simply requires tracing up the loop nesting forest from the block's loop reference. Finding the exit sets to which a block $b \in R$ belongs requires doing the same for each of b's predecessors (and ensuring containment for the candidate loop).

The Ramalingam and Reps dominator tree reconstruction algorithm [17] discussed in Section 6 specifies that the processing of blocks must be done in the same order as the block's priority ordering. Suppose that blocks a and b are both in the affected set and calculating the immediate dominator of a relies on knowing correct immediate dominator of b. In this case, processing block a before b may lead to incorrect results. This incorrect ordering does not occur if blocks are processed topologically. In the implementation of Waddle, we forgo tracking priority orders of blocks to simplify the addition of edges, but at an increased cost during edge deletion. Instead of processing blocks in order, we simply re-calculate the immediate dominators of all affected blocks until no more changes occur. This trade-off seems beneficial in general as the cost of maintaining priority order on edge insertion is relatively high, but the out-degree of basic blocks (and therefore the maximum number of siblings in the tree) is relatively low.

[2] $\|A\|^{\leftrightarrow}$ denotes the number of blocks in the set A plus the number of edges entering and the number of edges leaving A.

References

[1] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

[2] S. Alstrup and P. W. Lauridsen. A simple and optimal algorithm for finding immediate dominators in reducible graphs, 1996.

[3] S. Alstrup and P. W. Lauridsen. A simple dynamic algorithm for maintaining a dominator tree. 1996.

[4] S. Beamer, K. Asanović, and D. Patterson. The GAP Benchmark Suite. *ArXiv e-prints*, Aug. 2015.

[5] M. Braun, S. Buchwald, S. Hack, R. Leißa, C. Mallon, and A. Zwinkau. Simple and efficient construction of static single assignment form. In *Compiler Construction*, pages 102–122. Springer, 2013.

[6] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 274–284, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. . URL `http://doi.acm.org/10.1145/73560.73584`.

[7] L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: reducible flow graphs are exponentially larger. In *ACM SIGPLAN Notices*, volume 38, pages 106–114. ACM, 2003.

[8] E. Fritz. *Waddle – Always-Canonical Intermediate Representation*. PhD thesis, University of Wisconsin – Milwaukee, Milwaukee, Wisconsin, Expected 2018.

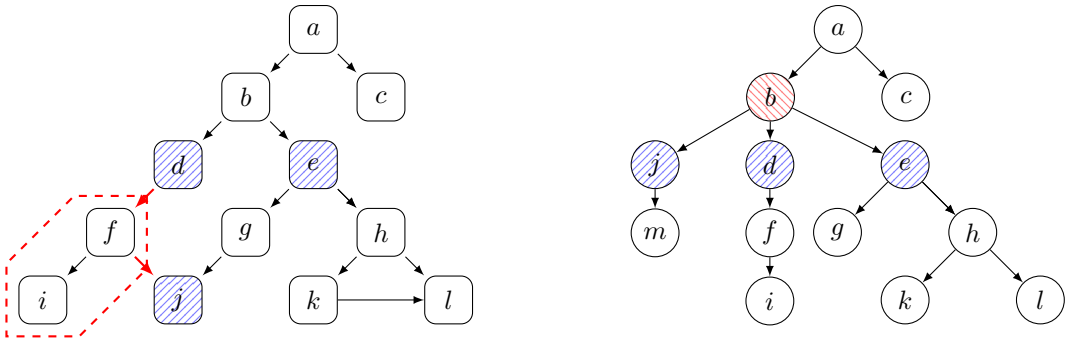


Figure 12: A flow graph (left) and its dominator (right) tree. The deletion of edge (d, f) will require additional processing of edge (f, j). The newly unreachable blocks are outlined. The set of possibly affected nodes (the siblings of j in the dominator tree) are shaded.

[9] L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. An experimental study of dynamic dominators. In *Algorithms–ESA 2012*, pages 491–502. Springer, 2012.

[10] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4): 557–567, 1997.

[11] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, Nov. 1997. ISSN 0164-0925. . URL `http://doi.acm.org/10.1145/267959.269971`.

[12] R. Johnson. *Dependence Based Program Analysis*. PhD thesis, Cornell University, Ithaca, New York, August 1994.

[13] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[14] N. Parotsidis and L. Georgiadis. Dominators in directed graphs: a survey of recent results, applications, and open problems. 2013.

[15] K. Patakakis, L. Georgiadis, and V. A. Tatsis. Dynamic dominators in practice. In *2011 Panhellenic Conference on Informatics*, pages 100–104. IEEE, 2011.

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

[17] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 287–296. ACM, 1994.

[18] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. An efficient incremental algorithm for maintaining dominator trees and its application to ϕ-nodes update. ACAPS Technical Memo 77, 1994.

[19] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental computation of dominator trees. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-89791-754-5. . URL `http://doi.acm.org/10.1145/202529.202531`.

[20] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, 1996.

[21] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. 1993.

[22] R. Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107. ACM, 1973.

[23] S. Unger and F. Mueller. *Handling irreducible loops: Optimized node splitting vs. DJ-graphs*. Springer, 2001.