Partial Control-Flow Linearization

Simon Moll
Saarland University
Saarland Informatics Campus
Germany
moll@cs.uni-saarland.de

Sebastian Hack
Saarland University
Saarland Informatics Campus
Germany
hack@cs.uni-saarland.de

Abstract

If-conversion is a fundamental technique for vectorization. It accounts for the fact that in a SIMD program, several targets of a branch might be executed because of divergence. Especially for irregular data-parallel workloads, it is crucial to avoid if-converting non-divergent branches to increase SIMD utilization. In this paper, we present partial linearization, a simple and efficient if-conversion algorithm that overcomes several limitations of existing if-conversion techniques. In contrast to prior work, it has provable guarantees on which non-divergent branches are retained and will never duplicate code or insert additional branches. We show how our algorithm can be used in a classic loop vectorizer as well as to implement data-parallel languages such as ISPC or OpenCL. Furthermore, we implement prior vectorizer optimizations on top of partial linearization in a more general way. We evaluate the implementation of our algorithm in LLVM on a range of irregular data analytics kernels, a neutronics simulation benchmark and NAB, a molecular dynamics benchmark from SPEC2017 on AVX2, AVX512, and ARM Advanced SIMD machines and report speedups of up to 146% over ICC, GCC and Clang O3.

CCS Concepts • Computer systems organization → Single instruction, multiple data; • Software and its engineering → Compilers; • Computing methodologies → Parallel programming languages;

Keywords SIMD, SPMD, Compiler optimizations

ACM Reference Format:

1 Introduction

Vectorization is an essential technique to achieve performance on data-parallel workloads on machines with SIMD instructions. Data-parallel workloads originate from dedicated data-parallel programming languages like OpenCL, CUDA or ISPC, but also from classic loop vectorization.

Consider the example in Figure 1 that shows the implementation of an element search in a binary tree. Assume that i is the thread index, i.e. the ID of the SIMD instance. (In the context of loop vectorization one would say that the body of the function is the loop body and i the induction variable of the loop.) The code returns the node index for each value Q[i] if the value is in the tree, and −1 otherwise.

This code is not straightforward to vectorize because it contains divergent (line 14) as well as uniform (lines 17 and 19) branches. A branch is called uniform if we can statically decide—by means of a divergence analysis [4, 8, 22]—if all SIMD instances will take it or not.

The common technique to handle divergence is control-flow linearization, also known as if-conversion. Thereby, all instructions that are affected by divergent branches are linearized into a single basic block and branching is replaced by predication to suppress illegal computations (see Section 2 for more background).

The problem with linearization is that SIMD utilization, and thus performance, drops because most of the time some

---

1 The uniform condition \( \text{any}(v) \) evaluates to true for all SIMD lanes, if \( v \) evaluates to true for any SIMD lane. Otherwise, \( \text{any}(v) \) is false.
instances are inactive. While linearization cannot be avoided on divergent control flow, it is absolutely mandatory to avoid linearization of uniform control flow to produce vector code that actually leads to speed ups for such kinds of workloads. For example, if we apply the algorithm we present in this paper to MPC—a data analytics kernel—we obtain a 7.31× speedup over scalar code. With standard if-conversion, the same benchmark times out after one hour.

The underlying problem is that existing linearization techniques either fully if-convert the CFG [2], require structured control flow [29], or contain other special cases and might create unwanted control flow artifacts [21]. If these requirements are not met, these algorithms fail to retain uniform edges, linearize code where not necessary, and therefore deteriorate SIMD utilization. There exist domain-specific vectorization approaches that are specific to certain problems, such as tree traversal [20, 30]. They perform very well in their particular domain but are not applicable in a general way.

A significant part of the benchmarks we consider in this paper has unstructured, mixed divergent/uniform control flow. Hence, standard if-conversion techniques fail to retain uniform control flow sufficiently. To the best of our knowledge, there is no technique that is able to reliably retain uniform control flow without making strong structural assumptions on the program.

In this paper, we present a novel if-conversion algorithm called partial linearization whose only requirement is reducible control flow, i.e., the absence of multi-entry loops which in practice almost all programs fulfill. Furthermore, our algorithm is simple, efficient, and, in contrast to previous approaches, provides strong, provable guarantees on the extent of the retained uniform control flow. On the benchmarks we consider, partial linearization retained all branches that were statically classified as uniform.

In summary, this paper makes the following contributions:

- We present partial linearization, a novel partial if-conversion algorithm (Section 3). Partial linearization is simple to implement and linear in the number of CFG edges. Unlike previous work, we prove our algorithm correct and provide proven criteria on the retained uniform control flow (Section 4).
- We show how the guarantees that partial linearization gives, allow for nicely incorporating dynamic techniques such as BOSCC [36].
- We implemented partial linearization in our vectorizer RV that vectorizes LLVM bitcode. We evaluate the implementation on a range of irregular data analytics kernels, a neutrinos simulation benchmark and NAB, a molecular dynamics benchmark from SPEC2017 on AVX2, AVX512, and ARM Advanced SIMD machines and report speedups of up to 146% over ICC, GCC and Clang O3 (Section 7).

2 Background

In this section, we recap basic definitions and review vectorizing data-parallel programs.

2.1 Prerequisites

A CFG $G = (V, E, entry)$ consists of basic blocks $v \in V$, control-flow edges $(b, i, s) \in E$ and a designated entry $v \in V$ such that every block $v \in V$ is reachable from entry. There is a terminator instruction at the end of every basic block. If the terminator is a branch they it has an array of successors. If $(b, i, s) \in E$ than $s$ is the $i$-th successor of the branch in $b$. Return instructions have no successors. We will use the notation $b \rightarrow s \in E$ to mean $\exists_i (b, i, s) \in E$. Likewise, we will use the notation $\pi \in a \rightarrow^* b$ to mean a path $\pi$ from $a$ to $b$ through a chain of edges. We call a path complete if its last block has no outgoing edges. The set $a\downarrow$ contains all complete paths that start in $a \in V$. We assume that $\forall a \in V, a\downarrow \neq \emptyset$, that is all loops have exits. We require that all edges back to loop entries originate in a single block, called the unique latch block. This can be achieved in reducible loops by merging all back edges.

In a graph $G$, the block $a \in V$ is said to dominate $b \in V$ ($a$ is a dominator of $b$), written $a \geq^0 b$, if every path $\pi \in entry \rightarrow^* b$ contains $a$. Symmetrically [9], the block $a \in V$ is said to post dominate $b \in V$ ($a$ is a post dominator of $b$), written $a \geq^{\pi_0} b$, iff every complete path $\pi \in b\downarrow$ contains $a$.

A block $k \in V$ is control dependent on an edge $a \rightarrow b \in E$, iff $k \geq^{\pi_0} b$ and $k \not\geq^{\pi_0} a$. We use the notation $cdep(k) \subseteq E$ to denote the set of all $a \rightarrow b \in E$ that $k \in V$ is control dependent on [9, 12].

2.2 Vectorization of Data-Parallel CFGs

We consider the program to be given by its control flow graph (CFG). In the data-parallel execution model, a CFG is instantiated for $N$ threads. These threads run in no prescribed order with a unique thread index. Data-parallel programs appear in inner as well as outer loop vectorization and in dedicated programming languages like OpenCL, CUDA or ISPC. To implement data-parallel programs on machines with explicit SIMD instructions (i.e., CPUs), a compiler has to vectorize the program accordingly. This is typically performed in four stages.

First, a static divergence analysis determines which variables are uniform. Informally, a variable is uniform if its value is equal among all threads. Non-uniform variables are called varying. A branch is called uniform if its branch condition is uniform, otherwise it is called divergent. An unconditional branch is always uniform. All threads that reach a uniform branch will take the same branch destination and therefore the branch might be retained in the vectorized program. A loop is called divergent if SIMD threads that enter the loop will leave it in different iterations or through different loop exits. Otherwise, the loop is uniform.
Second, instructions are inserted that compute the control predicate for every basic block. Third, if-conversion is used to eliminate divergent branches from the CFG. Finally, the vector code backend replaces every non-uniform instruction with a vector instruction. It also predicates instructions or inserts so-called blending code to mask out the results of the inactive threads.

### 2.3 Divergence Analysis

![Diagram](image)

**Figure 2.** (a) Function `run` with shapes (varying `v` and uniform `u`). (b) CFG with branch shapes (below blocks), edge predicates (light gray at edges) and block predicates (light gray inside blocks). (c) partially linearized CFG, control is uniform, block predicates have shapes.

Intuitively, a vectorized program executes the code of the scalar program for every SIMD thread in lockstep. As an instruction is executed, every SIMD thread produces an individual output for it. Divergence analysis [4, 8, 22] determines statically for each variable a *shape* that describes how the value of the instruction relates across SIMD threads.

*Figure 2a* shows an example of Whole-Function Vectorization [22]. The vectorizer will create a SIMD version of the scalar function `run`. In that vectorized function, the parameter `k` will be a vector, its shape in the analysis is thus *varying*. The parameter `n` will remain a scalar, and thus has a *uniform* shape. Divergence analysis propagates these initial shapes through the data flow graph to derive the shapes of all instructions. The inferred shapes are annotated as comments in Figure 2a. The if-statement in Line 8 is divergent since it transitively depends on the variable `k`.

For the purpose of if-conversion, we are only interested in the uniform and varying shapes of branch conditions. More elaborate shapes [8, 15] help for other optimizations. Divergent branches are if-converted for vectorization because SIMD CPUs can not handle divergent branches in hardware.

### 2.4 Predication

*Figure 2b* shows the CFG of Figure 2a and Figure 2c the result after if-conversion. In the original program (Figure 2a), line 9 may only execute if the condition `m` holds. Line 9 corresponds to the block `a` in the CFG of Figure 2b. If the CFG is if-converted, `a` will execute whenever the loop iterates. However, it is only safe to perform the load in `a` if the condition `q ∧ m` holds as indicated in Figure 2b.

To control the execution of basic blocks, the vectorizer predicates them. Whenever execution reaches a basic block the instructions in it perform their effect only if the predicate is true. The vectorizer inserts additional instruction in the blocks that compute the predicates.

Given a CFG `G`, the vectorizer generates predicates for all basic blocks `b ∈ V` and all edges `a → b ∈ E`. The predicate for an edge `a → b` is the conjunction of the block predicate of `a` and the branch condition of a leading to `b`. The predicate of a block `b` is the disjunction of the edge predicates of the control dependence edges of `b` [28].

The generated predicates have shapes as all other values in the program, shown in Figure 2c for the block predicates. In formal notation, we denote that a block has a uniform predicate by `uni(a)` for `a ∈ V`. We call an edge `a → b ∈ E` uniform, written `uni(a → b)`, iff `uni(a)` and block `b` ends in a uniform branch. If the constituents of a block predicate are all uniform, then the predicate of the block itself is uniform, i.e., `uni(a) ⇐⇒ uni(cdep(a))`.

### 3 Partial Linearization

In this section, we present a novel if-conversion algorithm that linearizes control flow only partially and retains certain uniform branches. We begin with an informal overview over the algorithm, prove its correctness, and finally prove two properties of our algorithm that characterize the uniform control flow edges it can retain.

#### 3.1 Block Index

A block index `Index : Blocks → {0, ..., n − 1}` is a topological sort of the basic blocks of a CFG (with backedges removed) that satisfies compactness constraints. A topological block sort `Index` is *compact* with respect to a set of basic blocks `B ⊂ Blocks`, iff all blocks in the range of

```
[min{Index(b) | b ∈ B}, max{Index(b) | b ∈ B}]
```

are also elements of `B`. In Figure 3a the blocks `c`, `f` and `g` fall compactly in the range `4` to `6` because `c` dominates them. A block `Index` is a topological block enumeration that is *compact* with respect to the element blocks of all loops and dominated-block sets [42]. We require *reducible* loops, which have a unique header that dominates all other nodes in the loop [17]. Unique loop headers have the minimum index of their loop’s blocks.
3.2 Algorithm

The algorithm, shown in Figure 5, works on loop-free CFGs. Because we require reducible CFGs, loop headers and back edges can be unambiguously identified. Hence, to get an appropriate CFG, we remove all backedges. Section 3.3 elaborates why the algorithm is still correct for CFGs with reducible loops.

The result of the algorithm is a new CFG $G_\ell = (V, E_\ell, \text{entry})$ that constitutes a partially if-converted version of the original graph $G = (V, E, \text{entry})$. Coming back to the example, the initial graph is shown in Figure 3a and the final graph $G_\ell$ in Figure 3f.

The algorithm visits every block in $V$ in block index order. At block $b \in V$, the algorithm creates outgoing control flow edges from $b$ and adds them to $E_\ell$, the set of edges in the resulting, if-converted CFG.

If block $b$ has a divergent branch, the branch needs to be if-converted and receives only a single outgoing edge in $G_\ell$. However, if a path in $G_\ell$ reaches the block $b$ then all of the original successor blocks of $b$ have to be part of every possible completion of that path. In other words, if the algorithm picks a successor next $\in V$ for $b$ in $G_\ell$ it has to make sure that all other successors of $b$ in the original graph will post-dominates $b$ in $G_\ell$ so that all successors will eventually execute.

To guarantee this, the algorithm maintains the deferral relation $D$. The algorithm ensures that whenever a pair $(v, w) \in V \times V$ is put into $D$, the node $w$ will end up post-dominating $v$ in $G_\ell$ (Lemma B.3 in the Appendix). When the algorithm visits a block $b$ with a divergent branch, it will put all the suspended original successors of $b$ into that relation. To make the deferral relation effective, the algorithm takes the elements of $D$ for the current node $b$ into account when picking a new successor for $b$.

3.3 Partial Linearization of Loops

Let us now discuss how to extend Figure 5 to support uniform, reducible loop nests. Section 5 discusses how reducible divergent loops can be converted into uniform loops. Hence, partial linearization does not have any other restriction than requiring reducible control flow.

Running Figure 5 on the CFG that has all backedges deleted is safe because of the following argument: We require the latch block to be unique (Section 2.1). It therefore has the maximum index of any block in the loop. Hence, the latch block is the only place to re-insert the backedge even in $G_\ell$. This is sound because all deferred edges of latch blocks lead outside the loop.

The deferral relation at the latch can only refer to blocks that were already deferred at the loop header. This is because uniform loops have no varying loops exits that could defer blocks that are outside of the loop. Therefore, if the latch is reached during execution of $G_\ell$ it is safe to assume that no exit from the loop was taken in this iteration. Thus, if the latch is not exiting itself, the latch can proceed with the next loop iteration.

Figure 8 shows how partial linearization deteriorates if the block index is not loop compact.

3.4 Correctness

Figure 5 is only concerned with producing a partially linearized CFG and relies on proper predication of the code.
Proof. There are two cases: Either $v = b$ or not.

First, assume $v = b$. $b$ certainly has a predecessor in $P$ because the nodes are visited in topological order, hence it fulfills the premise of the lemma.

Now, $b$ either ends in a uniform branch or not. Consider the first case. The inner loop (line 7) determines for each successor of $b$ (in $G$) one successor (next) in $G_F$. If next is picked to be $s$, then the edge $(b, s)$ is added to $G_F$ (line 9). If next is no successor of $b$ in $G$, the deferred edge from next to $s$ is added to $D$ in line 10. Hence, there is a path (in $E_F \cup D$) from $b$ to $s$.

If $b$ does not end in a uniform branch, a similar reasoning applies. Hence, the lemma also holds for all successors of $b$ that is added to $P$ at the end of the loop body.

Now, consider $v \neq b$. line 18 deletes deferred edges and we have to make sure that the invariant still holds for a node $v \neq b$. There could be a path $\pi$ in $E_F \cup D$ from some predecessor $u$ of $v$ in $G$ that contains an edge $(b, t)$ that is removed in line 18. However, in lines 10 and 16, all deferred edges that originate from $b$ are “re-originated” to next. Because the edge $(b, \text{next})$ is added to $E_F$, the to-be-removed edge $(b, t)$ can be replaced by the two-edge path $b, \text{next}, t$ in $\pi$. Hence the property is preserved for all other nodes unequal to $b$. \hfill $\Box$

**Theorem 3.2.** For each path $\pi$ of $G = (P \cup F, E)$, there is a path $\pi'$ in $G_F = (V, E_F \cup D)$, such that $\pi$ is a sub-path of $\pi'$.

**Proof.** By induction on $P$ (the outer loop). The base case trivially holds because $P \cup F$ is empty at the beginning of the program.

For the induction step, assume that the induction hypothesis holds for the subgraph of $G$ induced by the nodes in $\{b\} \cup P \cup F$. First of all, each predecessor of $b$ (in $G$) has already been processed because the nodes are processed (in the outer loop) in topological order. Hence, Lemma 3.1 applies to $b$.

Consider a path $\pi \in entry \rightarrow^* p$ in $G$ where $p$ is a predecessor of $b$. By the induction hypothesis, there is also a path $\pi'$ in $G_F$ that contains $\pi$ as a subpath. Consider the extension $\pi \circ (p \rightarrow b)$ of $\pi$ to $b$. By Lemma 3.1, there is either an edge $(p, b) \in E_F$ or a path $p \rightarrow^* b$ in $E_F \cup D$. \hfill $\Box$

The path embedding follows from the fact, that after the algorithm terminated, $P \cup F = V$ and $D = \emptyset$.

It remains to show that if both CFGs, original and partially linearized, are run with the same input values the original CFG will generate a trace that is embedded in the trace of the partially linearized CFG. Partial linearization never introduces new branches. Further, if partial linearization changes a branch target then the former branch target will post-dominate the new successor in the partially linearized CFG. In conjunction with Theorem 3.2 this means that any execution trace of the original CFG will also be part of the trace in the partially linearized CFG.

## 4 Guarantees

In this section, we prove two properties of partial linearization that characterize the uniform control flow that can be retained.

### 4.1 Preservation of Uniform Control Dependence

In an if-converted program, every instruction executes with a predicate unless the predicate is constant. Predication can
incur a significant performance overhead because predicates are computed and, even more severe, memory accesses and function calls need to be guarded, for example by additional branching. Therefore, it is desirable to avoid predicated execution where possible.

Partial linearization guarantees that predicates can be elided if the predicate of a block is uniform even if the predicate is non-constant. With this guarantee the code generator can safely emit efficient unpredicated instructions for basic blocks with uniform predicates. We make this guarantee precise in Theorem 4.1 and provide a proof.

**Theorem 4.1.** If $\text{uni}(b)$, i.e. the predicate of a block $b \in V$ is uniform, then execution will reach block $b$ in $G_\ell$ iff the predicate of $b$ is true.

The proof makes use of Lemma 4.2, which states that if $\text{uni}(k)$ then the control dependences of $k$ are preserved in $G_\ell$. We provide the proof for Theorem 4.1 here and refer the reader to Appendix B for a full technical proof for Lemma 4.2.

**Lemma 4.2.** If $\text{uni}(k)$ then $\text{cdep}(k) = \text{cdep}_\ell(k)$ where $\text{cdep}_\ell$ is the control dependence in $G_\ell$.

**Proof.** We now prove Theorem 4.1. We will first show that if $k$ is executed in $G$ then it is also executed in $G_\ell$. This follows from the correctness of partial linearization and that if $\pi$ is a path in $G$ with $k \in \pi$ then $\pi$ is embedded in a path $\pi'$ in $G_\ell$ with $k \in \pi'$.

It remains to show that if execution reaches the block $k$ in $G_\ell$ then block $k$ will also execute in G. We prove the claim by induction over the block index. Theorem 4.1 is the induction hypothesis.

**Base case:** If $\text{cdep}(k) = \emptyset$ then $k$ is always executed in $G$. Since every path in $G$ is embedded in a path in $G_\ell$, the block $k$ is also always executed in $G_\ell$. Note that $\text{cdep}(\text{entry}) = \emptyset$ for entry, the first block in the block index.

**Induction step:** Assume $\text{uni}(k)$ for some $k \in V$. We need to show that if $k$ is executed in $G_\ell$ then $k$ is also executed in $G$.

Let $\pi' \subseteq \text{entry} \rightarrow^* k$ be an arbitrary prefix path to $k$ in $G_\ell$. Then, there is an edge $a \rightarrow b \in \text{cdep}_\ell(k)$ such that $\pi' \subseteq \text{entry} \rightarrow^* a \rightarrow b \rightarrow^* k$.

By Lemma 4.2, $a \rightarrow b \in \text{cdep}(k)$ as well. Since $\text{uni}(k)$, it follows that $\text{uni}(\text{cdep}(k))$ and thus $\text{uni}(a)$ and the branch in $a$ is uniform.

By the induction hypothesis for $a < k$ it follows that $a$ will only be executed in $G_\ell$ if it is executed in $G$. Since the branch in $a$ is uniform this implies that the edge $a \rightarrow b$ will only be taken in $G_\ell$ if $a \rightarrow b$ is taken in $G$.

However, $a \rightarrow b \in \text{cdep}(k)$ implies that $k \geq^\pi b$ and thus any complete path in $G$ that contains $b$ will eventually pass through $k$. Hence, if $\text{uni}(k)$ and $k$ is executed in $G_\ell$ then it is executed in $G$ as well.

4.2 Preservation of Uniform Branches

Partial linearization preserves uniform branches in blocks with uniform predicates, as implied by Theorem 4.1. However, the algorithm will even preserve some uniform branches in blocks with varying predicates.

Figure 7 shows an example of this. Block $b$ has a uniform branch but its predicate is varying because $b$ is control-dependent on the edge $a \rightarrow b$, which is varying. Still, the uniform branch in $b$ will be preserved.

We present a branch preservation guarantee that extends to those branches as well. The guarantee uses the concept of relative uniformity of predicates. A block $b$ is uniform relative to its dominator $d$, if $b$ has only uniform control dependences in the dominance region of $d$. We will refer to the dominance subgraph of $d$ as $G^d$, formally defined by Definition 4.3.

**Definition 4.3.** The dominance region $G^d = (V^d, E^d, d)$ is the subgraph of $G = (V, E, \text{entry})$ that $d \in V$ dominates: $E^d = \{x \rightarrow y \in E \mid d \geq^\pi x\}$ $V^d = \{x \in V \mid d \geq^\pi x \land (\exists y, y \rightarrow x \in E^d)\}$

A block $b$ has a uniform predicate relative to a dominator $d$, if $b$ has a uniform predicate in the subgraph defined by the dominance region of $d$. This is formalized by Definition 4.4.

**Definition 4.4.** Let $d$ be a dominator of $b$. Consider the dominance region graph $G^d$ rooted in $d$. The entry mask of $d$ in $G^d$ is uniform. We call $b$ uniform relative to $d$, iff $b$ has a uniform mask in $G^d$.

In the example of Figure 7, we show the dominance region graph $G^b$ of $b$ in the center. The block $b$ dominates $c$ and so the edge $b \rightarrow c$ will be preserved. Generally, as stated by Theorem 4.5, if an edge $a \rightarrow b$ is uniform relative to
Theorem 4.5. Given a dominance-compact block index, partial linearization will preserve an edge \( b \rightarrow y \in E \) if \( \text{unit}(b) \) or there exists a block \( d \in V \) with the following properties in \( G \):

1. \( d \preceq b > 0 \), \( d > 0 \ y \) (\( d \) dominates the edge \( b \rightarrow y \)).
2. \( \text{unit}(b \rightarrow y) \) in the dominance region \( G_d \) of \( d \).

One non-obvious implication of Theorem 4.5 is that we can insert tests for all-false masks in the CFG (BOSCC) [38] even before if-conversion (Section 6). If the mask is all false, partial linearization guarantees that the guarded block and all blocks that it dominates will be skipped.

Proof We give an intuition why Theorem 4.5 is correct. The full proof can be found in the Appendix C. The insight behind the theorem is that partial linearization makes the same decisions on a dominance region as it does on the whole graph.

To this end, the block index of \( G \) has to be dominance compact. To see this, consider the non-dominance-compact block index in Figure 8. Block \( b \) dominates \( b \rightarrow d \) and \( b \rightarrow e \). However, as the unrelated block \( c \) is deferred at \( b \) and is next in the block index the uniform branch of \( b \) will be folded anyway.

5 Transforming Divergent Loops

Automatic vectorizers need to remove control divergence before code can be vectorized. To this end, divergent loops have to be turned into uniform loops.

In existing work, handling of divergent loops is usually spread out over the whole vectorizer pipeline [21, 40]. Hence, all stages have to consider the case that a loop could be divergent, as during if-conversion, mask generation and vector code generation.

We transform divergent loops into uniform loops by folding divergent exits into data flow. The transformed loops are still scalar but do not diverge through their loop exits.

Figure 8. Top: Effect of non dominance-compact block index. Bottom: Effect of non loop-compact block index. Left: original CFGs \( G \) with (non compact) block index, Center: processed up to 1, Right: \( G_r \) with defect.

In our setting, all data flow is in SSA form. \( \phi \)-nodes select incoming values depending on the predecessor block that reached them. If a predecessor edge is if-converted, \( \phi \)-nodes are replaced with blend instructions that switch on the predicates of the folded edges to pick a value [16].

Figure 9. Inner loop of Mandelbrot with a kill exit (for condition) and a divergent exit (if condition). \( z \) is varying. Limit and ESCAPE are uniform.

Figure 10 shows the inner loop of a Mandelbrot set computation. Figure 10 shows the corresponding CFG on the left. The loop runs for every pixel of an image with varying values of \( z \) for each pixel. The loop exit in Line 4 is divergent because in every iteration some SIMD threads may exit the loop here while others continue. Thus the Mandelbrot loop is divergent as a whole. The iteration variable \( i \) is used outside of the loop. For every thread, the value of \( i \) is the number of the iteration when the thread exited the loop. Since the loop trip count varies by the thread, \( i \) is varying, too.

The divergent loop transformation will transform the Mandelbrot loop into the uniform loop shown in the center of Figure 10. Thereby it operates in two stages:

First, the transformation creates a live mask \( \phi_{\text{liveMask}} \) node in the loop header to track the live threads in the loop. For each exit to a block \( x \), another mask \( \phi_{\text{ExitMask}} \) node is added to the loop header to record which thread left the loop to the exit \( x \). In the example, these are the exits to \( f \) and \( g \) and so there are \( \phi_{f\text{ExitMask}} \) and \( \phi_{g\text{ExitMask}} \). The transformation will also create an empty loop latch block, called the pure latch block. That is block \( d \) in the example. Figure 11 shows the contents of the final pure latch \( d \).

The transformation inserts the only exit branch of the transformed loop in the pure latch. The branch continues with the loop header if any thread continues with the loop. As soon as this condition does not hold anymore, the branches exits the loop to a new dedicated exit block \( e \). That exit block
$\phi_{\text{init}} \leftarrow \{ \phi_{\text{init-b}}, b \}; [0, a], [0, a]$

$\phi_{\text{init}} \leftarrow \{ \phi_{\text{init-b}}, b \}; [\phi_{\text{init-b}}, c], \{ \phi_{\text{init-b}}, a \}$

$\phi_{\text{init}} \leftarrow \{ \phi_{\text{init-b}}, b \}; [\phi_{\text{init-b}}, c], \{ \phi_{\text{init-b}}, a \}$

$\phi_{\text{Out}} \leftarrow \{ \phi_{\text{Out-b}}, b \}; [\phi_{\text{Out-b}}, c], [\phi_{\text{Out-b}}, a]$

1 for (k = 0; k < n; k++) {
2 \ldots j = \text{pearlist[]} [k]; \ldots
3 xij = xi - x[\text{dim} \times j]; \ldots
4 r2 = xij \times \ldots
5 \text{if } (r2 > \text{rgbmaxpsmax2}) \text{ continue; } // 0 \%
6 \ldots sj = fs[j] \times (\text{rborn}[j] - \text{BOFFSET}) \ldots
7 \text{if } (\text{dij} > \text{rgbmax} + sj) \text{ continue; } // 0 \%
8 \ldots
9 \text{if } ((\text{dij} > \text{rgbmax} - sj)) \{ \ldots \} // 35.1 \%
10 \ldots \text{else if } (\text{dij} > 4.0 \times sj) \{ \ldots \} // 91.3 \%
11 \ldots \text{else if } (\text{dij} > \text{fabs}(ri - sj)) \{ \ldots \} // 75.0 \%
12 \ldots \text{else if } (\text{dij} > \text{fabs}(ri - sj)) \{ \ldots \} // 100 \%
13 \ldots \text{else if } (ri < sj) \{ \ldots \} // n/a \%
14 }

Figure 11. pure latch block (d) with mask update $\phi$.

$e$ will branch on the exit masks to dispatch all threads to their actual loop exit destinations ($f$ and $g$). Since there is only one uniform exit in the transformed loop from the pure latch $d$ to the dedicated exit block $e$, the loop is now uniform. The if-cascade dispatching to the original loop exits $f$ and $g$ potentially contains divergent branches. However, these are now part of the parent loop.

Second, the divergent loop transform rebounds every exiting branch to jump to the pure latch instead of the original loop exit. When a rebound edge is taken, the loop live mask and the loop exist masks are updated with additional $\phi$ nodes in the pure latch block. The node $\phi_{\text{live}}$ sets the live mask to zero if the latch is reached from any rebound exiting edge and maintains the old live mask otherwise. The nodes $\phi_{\text{exit}}$ update the exit masks for blocks $f$ and $g$.

If the pure latch is reached from a former exiting block, the live mask is set to 0 and the exit mask to the predicate of the exiting edge. In Figure 10, the exit from $a$ is rebound to the pure latch $d$. The former latch block $b$ also had an exiting edge. We break the exiting edge of the former latch block by inserting a new block $c$. Its only purpose is to have a non-exiting incoming edge form $b$ to update the $\phi$ nodes.

We insert an any mask intrinsic in the pure latch to check whether any thread will continue in the loop and exit to block $e$ otherwise. Partial linearization will regard it as a regular uniform branch. The backend lowers the intrinsic, for example with a ptest instruction on x86 AVX2 targets.

**Partial Linearization of Transformed Loops** When the transformed loop is visited during partial linearization the uniform edge from $a$ to $d$ will be retained. The rebound divergent branch from $b$ to $g$ will be if-converted. The resulting CFG is shown on the right of Figure 10. The $\phi$-nodes will be folded down to blends (not shown here).

6 BOSCC with Partial Linearization

Branch on Supercondition Code (BOSCC) [36] is a technique to add dynamic tests for uniformity to skip over linearized code for which a static analysis failed to prove uniformity. BOSCC inserts branches that skip a region if the predicate of the region entry evaluates to false for all SIMD threads. In this section, we show how to obtain BOSCC-ed code generically using partial linearization. By exploiting the guarantees we established in Section 4, we show that handling BOSCC is contained as a special case in partial linearization by adding a “BOSCC gadget” (see below) to the CFG before linearization.

Potential for BOSCC occurs in real benchmarks and applications. Consider the innermost hot loop from 644.nab_s benchmark from SPEC2017 shown in Figure 12. The dominating control feature of the loop is a deep if-cascade with very biased branch probabilities, shown as comments in Figure 12. For the three if-statements from Line 10 to Line 12 the probability to branch to the if-case is each at least 75% and even 100% for Line 12. So, there is a 91.3% chance that the loop will continue to the next iteration already after Line 10.

The if-branches in Figure 12 are divergent since they depend on the iteration variable $k$ and will be fully if-converted. This leads to inefficient SIMD code as the statements below Line 10 will often execute with an all false predicate. BOSCC branches placed at the if-else cases skip the remainder of the cascade as the predicate becomes all false. In fact, using BOSCC in Figure 12 leads to a speedup of 35% over the Intel C Compiler (icc) on AVX512.
6.1 The BOSCC Gadget

Consider the CFG in Figure 13a and suppose we want to insert a BOSCC-branch to skip block \( b \) if its mask is all false. Block \( b \) has the unique predecessor \( a \). We insert a BOSCC gadget, a small CFG pattern that makes partial linearization skip over \( b \) and its dominance region if its mask is all false. Figure 13b shows the installed BOSCC gadget.

The BOSCC gadget consists of a new block \( \text{any}(b) \) that contains the instructions of the original block \( a \) minus its terminator. The block gets a new uniform branch that jumps to \( a \), if any thread in the mask of \( b \) is true, and branches to \( c \) otherwise. The BOSCC gadget makes sure that \( b \) will only execute iff the predicate of \( b \) contains at least one live thread.

Figure 13c shows the CFG after partial linearization has passed through the BOSCC gadget. The divergent branch of block \( a \) has been if-converted while the \( \text{any}(b) \) branch persists as it is uniform. The linearized CFG will skip block \( b \), and its dominance region, if the predicate of \( b \) is all false. This is guaranteed by the branch preservation property (Theorem 4.5) of partial linearization.

In the hot loop of the nab benchmark, we insert all-false tests in three locations. On the left of Figure 13, we show the part of the CFG with the last four if-else cases (Lines 10 to 12) in the loop body. We insert three BOSCC gadgets to skip the if-statements contained in the else-cases, resulting in the CFG of Figure 13e. Figure 13f shows the linearized CFG. The locally-inserted BOSCC gadgets have a non-local effect on partial linearization; the order of the if-cases in the linearized CFG is reversed compared to the code of Listing 12. This arrangement lets the linearized CFG skip the remainder of the if-cascade as soon as one of the all-false tests succeeds.

7 Evaluation

We implemented partial linearization in RV \(^2\), a whole-function and outer-loop vectorizer for LLVM. Our implementation is based on the compiler framework LLVM 4.0.1 [25]. We evaluate our approach on a range of irregular workloads from data analytics benchmark suite, a neutronics simulation code and the 644. nab_s benchmark of SPEC2017 [39].

All experiments were conducted on an Intel 7900X CPU (Skylake) with AVX512 (512bit SIMD registers), an Intel Xeon E3-1225 CPU (Haswell) with AVX2 (256bit SIMD registers) and a Raspberry Pi 3 (ARM Cortex-A53 CPU) with Advanced SIMD (128bit SIMD registers).

In our case studies, we compare against the Intel C Compiler (ICC, 17.0.4), GCC (7.2.0) and Clang (4.0.1).

7.1 Irregular Data Analytics Kernels

These kernels are rich in unstructured control flow as well as uniform and divergent branches and have been found hard to vectorize [20].

---

\(^2\)https://github.com/cdl-saarland/rv

---

**Programming Model** The kernels are written as functions in scalar C++ code and make use of predicate intrinsics (popcount, any) to branch on properties of the predicate (number of live threads, etc). In scalar execution, these intrinsics are inlined and behave as if the vector width was 1.

**Benchmarks** We adopted the Vantage Point, Nearest Neighbor, Point Correlation, k-means clustering and Barnes-Hut data analytics kernels and data sets from the existing Lones- tar [24] and Treelogy [18] benchmark suites and added two new benchmarks: multi-radius point correlation (mpc) and binary tree (bt). To make the kernels amenable to vectorization, we replaced their recursive implementation by an explicit stack. Furthermore, we added a speculative traversal technique [1], a well-known technique to increase SIMD utilization for such codes. The following list describes the benchmarks and their input sets in further detail:

- **Barnes Hut (bh)** Acceleration structure for n-body simulations. random: 1000,000 random bodies. plummer: 100,000 bodies from a plummer model.
- **Vantage Point (vp)** Nearest Neighbor on a Vantage point tree. Same inputs as nn.
- **Point Correlation (pc)** Point correlation kd-tree implementation. Count the number of points that lie within a radius of a sample position. (Varying sample coordinates, uniform radius (50)). Same inputs as vp.
- **Multi-Radius Point Correlation (mpc)** Multi-radius point correlation [14]. Same inputs as PC.
- **K-means (km)** KD-tree based k-means algorithm (K = 128). Same inputs as nn.
- **Binary tree (bt)** Element search on a binary tree. random: 262,144 random elements.
- **XSBench binary search (xs)** Binary search in sorted array for maximal element below a query. This is the inner-most loop of the XSBench benchmark [41]. random: 4,194,304 elements.

**Multi-Radius Point Correlation** For the bh, nn, pc, vp and km benchmarks, the query coordinate is always varying while all other parameters to the query are uniform. It has been noted [14] that some machine learning applications benefit from a SIMD version of Point Correlation that takes a vector of radii and a single coordinate. Using our approach, we can automatically create such a SIMD kernel from the normal Point Correlation source code simply by changing the parameter shapes. The multi-radius point correlation kernel (MPC) is a point correlation kernel with a uniform coordinate and varying radii.
**Query Inputs** The PC, VP, NN, MPC, XS and bintree kernels query a data structure at user-specified coordinates. We draw uniform random coordinates from the bounding boxes of the data set. In case of bintree, we take 40% random samples from the data range with 50% chance of being a tree element. This array is then sorted. For the XSbench binary search, we draw \(2^n\) random samples and sort them. All versions of the kernels were run with the exact same inputs and query order. Performance differences are therefore due to vectorization and the employed if-conversion technique.

**Results** We evaluate the data analytics kernel under the following settings:

- **Partial linearization.** vectorized with partial linearization and divergent loop transform.
- **If-conversion.** vectorized with if-conversion and divergent loop transformation. If-conversion is the standard technique [2, 40] to eliminate divergent branches.
- **Baseline.** Scalar kernels compiled with O3 optimization level (includes LLVM’s loop and SLP vectorizers).

Note that our goal is generic vectorization of CFGs. Therefore, we do not compare against prior work on dedicated automatic vectorization of tree traversals [20] that achieves even better results but is limited to this particular kind of code and are not applicable to other codes such as 644.nab_s.

We show the results in Figure 14c for AVX512, in Figure 14f for AVX2 and in Figure 14i for ARM Adv. SIMD. Each figure shows the measured speed up over the Baseline on top and the average SIMD utilization below. The slowest partially linearized kernel finished within two minutes on AVX2 and AVX512 and within 45 minutes on Adv. SIMD. The timeout for AVX2 and AVX512 was thus set to one hour for AVX2 and AVX512 and to two hours on ARM Adv. SIMD. Timed out results are marked with an asterisk (*) and do not factor into the reported means. The average SIMD utilization is the average number of active SIMD threads per basic block execution divided by the vector width.

**Comparison with If-Conversion** Partial linearization outperforms if-conversion on all three machines and on all analytics kernels, except for xs. The xs kernel is extracted from the XSbench benchmark, which will be discussed in Section 7.3.

The SIMD utilization improvements are due to preserved uniform branches as Table 1 reveals (column Branch/pres).

In the if-converted kernels, all uniform branches are folded. This causes the blocks that the branch would otherwise skip to execute with an all-false predicate, which in turn drains SIMD utilization. This includes uniform top-level branches. However, the vector code backend will re-introduce the folded branches to guard instructions with side effects. LLVM will often merge and hoist these checks. The runtime numbers we report for the full if-conversion case include the full LLVM O3 pipeline run after our vectorizer.
The higher the number of uniform branches in the kernel (column Branch/pres), the more pronounced is the utilization gap between partially linearized and fully if-converted kernels. This effect is strongest for the mpc kernel that shows a 7.31 speed up with partially linearization but times out if full if-conversion is employed: mpc contains the most uniform branches and uniform loops of all of the benchmarks.

The memory accesses in all kernels, except for xs, operate either on uniform pointers or access contiguous memory (as in C[tid]). The xs kernel contains a single load from a varying pointer (a gather to load from the array) with a varying predicate (last column of Table 1). There is no gather instruction in the ARM Advanced SIMD ISA and the load is scalarized to an if-cascade. The same happens for AVX2 because, although gather instructions exist on AVX2, LLVM will not emit them for Haswell as they are deemed inefficient.

Comparison with Scalar Baseline On AVX512, all kernels show a speedup except for the pc benchmark with the covtype dataset. For the covtype dataset, the query radius is less then 5% of the diameter of the dataset (bounding box). Therefore, the redundancy gains by traversing the tree in lock step are low. This reflects in the low SIMD utilization of little more than 50% and thus translates to poor performance.

On the other hand, the mpc kernels achieve significant speedups over the scalar baseline. There is a single query coordinate for all SIMD threads in mpc. Therefore, the set of nodes to visit is highly redundant among the query instances grouped together by vectorization.

The evaluation machines cover three different vector widths from 128 Bits to 512 Bits. The SIMD utilization for a given benchmark is stable independent of the machine. Across the benchmarks the performance of the vectorized tree kernels scales with the vector width. In mpc and bt the performance scales roughly by a factor of 2 with the vector width, which is the theoretical maximum gain to be expected by a doubled vector width if microarchitectural differences are ignored.

### 7.2 Case Study: 644 nab_s

We use the 644.nab_s benchmark of SPEC2017 to show the efficacy of the BOSCC gadget. We evaluated on the SPEC2017 refspeed data set for AVX512/AVX2 and on the refrain data set for Adv.SIMD because of memory constraints. We compare against Clang (with PGO), GCC and ICC (on x86 only) as shown in Figure 14a, Figure 14d and Figure 14g.

About 77% of the running time in the 644.nab_s is spent in three hot loops of the egb function (amino profile). We will refer to these loops by the order they occur in the code (loops 1 to 3). We applied RV to all three loops with the full vector length of the target. We measured the time spent in each of these loops and the total running time on the benchmark. The first and third loop have the deep, divergent if-cascade as outlined in Figure 12.

None of the compilers (ICC, GCC, Clang) perform automatic loop vectorization on the three hot loops. Vectorization without BOSCC leads to regressions compared to Clang on all but the AVX512 target. BOSCC enables significant speedups for this benchmark on AVX512 and Adv. SIMD. We attribute the performance difference between AVX512 and AVX2 to the factor-two difference in vector width.

RV inserts three BOSCC gadgets in the first and third loop as shown in Figure 13. These branches add to the preserved branches in Table 1 for the */bsc variants of loops 1 and 3.

### 7.3 Case Study: XS Bench

XS Bench is a proxy benchmark for the key computational kernel of the Monte Carlo neutronics application OpenMC [33]. About 85% of the total runtime of the actual OpenMC application is spent in this code [41]. We run XS Bench with the nuc1.de grid type option. The input sizes were XL for AVX2 and AVX512 and small for Advanced SIMD due to memory constraints. We apply RV to an outer loop that internally runs the xs kernel as part of the simulation code. As shown in Table 1 the vectorization of that loop requires the divergent loop transform and partial linearization to preserve the uniform loop. Our approach attains a speed up of 146%

### 7.1 Comparison of Loops

Table 1. Branch, loop, loop exit and mask statistics. div branches are divergent branches, lost branches are uniform branches that had to be if-converted and pres branches are preserved uniform branches. uni/div are uniform/divergent loops (loop exits). true is the number of loads/stores with a constant true predicate, uni is the number with non-constant uniform predicates and var is for varying predicates.

<table>
<thead>
<tr>
<th>Name</th>
<th>Branch</th>
<th>Loop</th>
<th>Exit</th>
<th>L/S Masks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>div</td>
<td>lost</td>
<td>pres</td>
<td>uni</td>
</tr>
<tr>
<td>bh</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>bt</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>km</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>mpc</td>
<td>4</td>
<td>0</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>nn</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>pc</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>vp</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>xs</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Branch</th>
<th>Loop</th>
<th>Exit</th>
<th>L/S Masks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>div</td>
<td>lost</td>
<td>pres</td>
<td>uni</td>
</tr>
<tr>
<td>nab-1/vec</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>nab-1/bsc</td>
<td>7</td>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>nab-2/vec</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>nab-2/bsc</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>nab-3/vec</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>nab-3/bsc</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>xsbench</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The memory accesses in all kernels, except for xs, operate either on uniform pointers or access contiguous memory (as in C[tid]). The xs kernel contains a single load from a varying pointer (a gather to load from the array) with a varying predicate (last column of Table 1). There is no gather instruction in the ARM Advanced SIMD ISA and the load is scalarized to an if-cascade. The same happens for AVX2 because, although gather instructions exist on AVX2, LLVM will not emit them for Haswell as they are inefficient. Only for AVX512 does LLVM generate a gather instruction leading to the situation that this is the only target where speedups over scalar can be observed for the xs kernel.
(AVX512) and 14.24% (AVX2) over the best of GCC, Clang and ICC.

7.4 Partial Linearization

Table 1 shows that partial linearization preserves all uniform branches (column Branch/lost) across all benchmarks. Not a single uniform branch was folded (lost) as the byproduct of if-converting a divergent branch.

**Comparison with ISPC** The nab benchmark in the setting RV+BOSCC, shown in Figure 13, uses unstructured control flow. ISPC’s if-conversion technique isn’t applicable here. Transforming the CFG in order to make it structured would render the BOSCC gadget ineffective. The adopted Treelogy benchmarks use mixed uniform/varying short-circuit conditionals, as in if (U && V). ISPC defaults to full if-conversion in this case. Partial linearization will naturally preserve the branch on U and only fold the branch on V.

8 Related Work

There exist numerous optimizations to make data analytics kernels amenable to GPU execution [13, 19] and vectorization [20]. Data analytics kernels feature a mixture of uniform branches, for traversing the data structure, and divergent branches making these kernels hard to vectorize [20]. Therefore, Automatic SIMD vectorizers for traversal algorithms are highly specialized for this problem class [20, 30–32].

Uniform branch preservation has also been studied in the context of GPUs kernels [10, 27]. Preserved uniform branches make the GPU kernels more efficient. GPUs support divergent branches in hardware, which is why these works do not address if-conversion at all. However, eliminating divergent branches in the program is a strict requirement for SIMD CPUs. If-conversion is the principal technique to eliminate divergent branches for SIMD vectorization [2].

The Intel SPMD Program Compiler (ISPC) [29] operates on fully structured ASTs. As such, unstructured branches either need to be uniform (gotos) or will be if-converted completely. However, unstructured control flow appears in practice. For example, Bahmann et al. [6] showed that in SPEC2006, 4390 of 14321 CFGs are unstructured. Partial linearization subsumes ISPC’s if-conversion because partial linearization preserves all the branches that ISPC preserves. This follows as a corollary from Theorem 4.5. Hence, partial linearization is more powerful than ISPC’s heuristic.

The early algorithm by Ferrante and Mace [11] has an $O(n \log n)$ complexity and inserts blocks and branches. Karrenberg [21], Karrenberg and Hack [23] present an incomplete partial linearization algorithm that recovers control with additional (cluster-dependent) branches. These branches can cause irreducible control even if the original CFG was acyclic. For example, Karrenberg’s method already creates an irreducible loop for the CFG in Figure 3. Regarding compile time, partial linearization has linear complexity in the number of edges while Karrenberg’s method is quadratic and spans over five algorithm listings. For absence of guarantees the BOSCC-gadget would not reliably work with Karrenberg’s method.

A different class of algorithms insert new basic blocks, predicates and branches after complete if-conversion [3, 26, 37]. None of the aforementioned techniques gives comparable branch preservation guarantees to partial linearization.

Previous work has looked into handling loops with divergent exits. This includes the set up [22, 40] of live masks for divergent exits. Uniform exits in divergent loops were studied previously [23]. However, all of these approaches handle divergent loops specially throughout the vectorizer pipeline. Our approach makes divergent loops uniform in a standalone transformation. The following analyses and transformations, including the if-conversion algorithm, become simpler since all loops they see are uniform.

The BOSCC technique [36, 37] inserts BOSCC branches after if-conversion and requires a predicate hierarchy graph. Techniques related to BOSCC in GPU kernel optimization support BOSCC before if-conversion but only on SESE regions [27]. In contrast, the BOSCC gadget encodes the semantics of BOSCC branches directly in the CFG. Partial linearization then natively folds these down to their intended effect, even in unstructured control scenarios and without additional data structures [36].

Several techniques have been proposed to enable the loop vectorization of non data-parallel loops [5, 35]. The techniques presented here are applicable after these techniques have established the legality of vectorization. Techniques such as block unification [8, 34] that improve the utilization in divergent code are complementary to partial linearization.

9 Conclusion

In this paper, we presented partial linearization, a simple and efficient if-conversion algorithm for unstructured CFGs that focuses on retaining uniform control flow. Partial linearization can be used in a classic loop vectorizer as well as to implement data-parallel languages such as CUDA, OpenCL, or ISPC on a machine with explicit SIMD instructions. In contrast to prior work, partial linearization has provable guarantees on the extent of uniform control flow that can be are retained. At the same time, it will never insert new branches or duplicate code. We evaluate the implementation of our algorithm on a range of control-flow intensive kernels on which classical vectorizers fail to achieve speed ups. Partial linearization was able to retain all uniform branches in these benchmarks. On wide range of vector machines (AVX2, AVX512, ARM Adv. SIMD) we report speedups of up to 146% over ICC, GCC and Clang O3.


A Extended Notation & General Remarks

A.1 Extended Notation

The set of blocks that \( k \in V \) is control dependent on is defined as \( cdep(k) = \{ a \in V \mid \exists b.a \rightarrow b \in cdep(k) \} \).

We write \( \succsim \) and \( cdep_\ell \) to refer to post dominance and control dependence on the partially linearized graph \( G_\ell \).

We use the notation \( x@q \) for \( q \in V \) and \( x \) being a variable in the algorithm to refer to the value of variable \( x \) after its update in the outer loop iteration of block \( q \). For example, \( next@p \) is the value of variable \( next \) after line 14, if \( p \) has a varying branch. If \( p \) has a uniform branch than \( next@p \) refers to the value of \( next \) after line 8. In case of uniform branches there can be multiple definitions of \( next \) for \( next@b \).

The inner loop iteration \( next@b \) is referring to will be made clear in the context.

A.2 General Remarks

Note that line 18 can be removed from the algorithm without any effect on the resulting \( G_\ell \). This is because \( D@b \) is only read in the definitions of \( T@b' \) with \( b' > b \). Further, line 18 is the only statement that removes entries from the deferral relation. Thus, after a new pair \((x,d) \in D@b \) is added in line 10 or line 16, it will be the case that \( d \in T@x \).

B Preservation of Uniform Control Dependence

Lemma B.1. If \( \mathit{uni}(k) \) then \( cdep(k) = cdep_\ell(k) \) where \( cdep_\ell \) is the control dependence in \( G_\ell \).

It is the purpose of this Section to prove Lemma B.1 that was used as an unproven lemma in the proof of Theorem 4.1.

B.1 Auxiliary Lemmas

Lemma B.2.

\[ c \in T@b \implies \forall (b,s) \in E_\ell \{ (s,c) \in D@b \vee s = c \} \]

Note that \( T@b \) contains the deferral targets of \( b \) before \( D \) is modified while \( D@b \) includes the updates to \( D \) after the outer loop iteration for \( b \) has finished.

Proof. For any such \( c \in T@b \), we distinguish three cases in the outer loop in the iteration of \( b \in V \):

- Case 1. \( b \) has a divergent branch and \( x = \min(T@b) \) with \( \forall s < S@b.x < s \). Since \( x \leq \min(S@b \cup T@b) \) always \( next@b = x \). If \( x = c \) then \( (b,0,c) \in E_\ell \). Otherwise, if \( x \neq c \), then \( (b,0,x) \in E_\ell \) and \( (x,c) \in D@b \) after line 16.

- Case 2. \( b \) has a divergent branch and \( s = \min(S@b) < \min(T@b) \). So, \( next@b = s \) and \( next@b \notin T@b \). We get \( (b,0,s) \in E_\ell \) and \( (s,c) \in D@b \) because \( next@b \notin T@b \).

- Case 3. \( b \) has uniform branch.

For every iteration of the inner loop, there are two cases for each \( (b,i,s) \in E \): If \( next@b \neq c \) then \( (b,i,next@b) \in E_\ell \) and \( (next@b,c) \in D@b \) since \( c \in T@b \) and \( c \neq next@b \). Otherwise, if \( next@b = c \) then \( (b,i,next@b) \in E_\ell \).

Lemma B.3. \( c \in T@b \implies c \succsim \)

Proof. Given that \( c \in T@b \), consider every complete path \( \pi \in b \) in \( G_\ell \). Since \( \pi \) is complete it ends in some \( x \in V \) where \( x \) is a block without successors in \( G_\ell \). When the outer loop processed \( x \), it also held that \( T@x = \emptyset \). However, when \( b \) was processed it held that \( c \in T@b \). Hence, there must be a node \( m \in \pi \) where \( next@m = c \). To see why, assume that there was no \( m \in \pi \) with \( next@m = c \). By Lemma B.2, it must therefore hold that \( c \in T@x \). However, this contradicts that \( x \) has no successors in \( G_\ell \). As this reasoning applies to any complete path \( \pi \) from \( b \) in \( G_\ell \), the node \( c \) is element of any such path \( \pi \). Thus, by definition of post dominance, \( c \succsim \).

Lemma B.4. \( a \succsim x \implies a \succsim x \)

Proof. We show the claim by induction over the post dominance relation in \( G \).

Base case The claim trivially follows for \( a = x \).

Induction step Assume that \( a \succsim x \). For every successor \( p \) with \( x \rightarrow p \in E \) it holds that \( a \succsim p \). By the induction hypothesis therefore \( a \succsim p \). For every edge \( (x,i,next@x) \) in \( E_\ell \) there are two cases: Either immediately \( next@x \) is not \( p \) or it holds that \( next@x \neq p \). In the latter case \( next@x,p \) is added in \( D@x \) after the update to \( D \) and so \( p \succsim \) by Lemma B.3 with \( a \succsim x \). Therefore, in general \( a \succsim x \).

Lemma B.5. \( \mathit{uni}(a) \implies T@a = \emptyset \)

Proof. We will prove this claim by an outer induction over the block index and an inner induction over the post dominance region of a node. For the outer induction, the induction hypothesis is equivalent to the claim \( \mathit{uni}(a) \implies T = \emptyset \).

Outer base case For the first node in the block index, the claim follows from the initial state with \( D = \emptyset \).

Outer induction step We may assume that given \( \mathit{uni}(a) \) it holds that \( \forall d \in cdep(a).T@d = \emptyset \). This is because \( \mathit{uni}(a) \) implies \( \mathit{uni}(cdep(a)) \). It remains to show that then also \( T@a = \emptyset \). We will prove this by induction over the post dominance region of \( a \) in block index order. The induction hypothesis for the inner induction step is \( a \succsim p \implies (\forall t \in T@p.a \succsim t) \). For the case that \( p = a \), this implies that \( T@a = \emptyset \) because \( \forall t \in T@a.t > a \).

Inner base case The base case for the inner induction is the minimum node \( p \in V \) with \( a \succsim p \). If \( T@p = \emptyset \) the claim follows trivially. Otherwise, assume there exists a \( t \in T@p \).

First, note that \( p \notin T@x \) for any \( x \in V \). Assume that \( p \in T@x \), there must be a node \( s \) with the edge \( s \rightarrow p \in E \) during which processing the pair \( next@s,p \) was inserted into the deferral relation. Then, \( a \succsim s \) because \( p \) is the minimum node with \( a \succsim p \) and hence \( s \rightarrow p \in cdep(a) \).
With $\text{uni}(a)$ it follows that $s$ has a uniform branch and the outer induction hypothesis implies that $T@s = \emptyset$. Therefore, always $(\text{next}@s,p) \notin D@s$ after line 10, for any such $s \to p \in E$. This contradicts $p \in T@x$ for any $x \in V$.

So, if $t \in T@p$ due to $(\text{next}@q,t) \in D@q$ with $\text{next}@q = p$ then $q \to p \in E$. However, then again $q \to p \in \text{cdep}(a)$ and $q$ must have a uniform branch and the outer induction hypothesis yields $T@q = \emptyset$. Thus, $(\text{next}@q,t) \notin D@q$ after the outer loop has finished processing $q$. Therefore, $t \in T@p$ cannot exist and finally $T@p = \emptyset$.

**Inner induction step** We proceed with the inner induction step for a node $p \in V$ such that $a \geq_{\varnothing} b$. Again, consider there was a $t \in T@p$ such a $\not\exists_{\varnothing} t$ while $a \geq_{\varnothing} p$. There must have been an outer loop iteration of the algorithm for a node $s \in V$ (i.e. $b = s^k$) such that $\text{next}@s = p$ and $(p,t) \in D@s$ after the iteration.

We distinguish three cases for $s$:

- **Case 1.** $s \to p \in \text{cdep}(a)$. Therefore $s$ has a uniform branch and by the (outer) hypothesis it holds that $T@s = \emptyset$. This leads to the contradiction that $(p,t) \notin D@s$ after $s$ was processed.

- **Case 2.** $a \geq_{\varnothing} s$. As $s < p$, we can apply the inner induction hypothesis and obtain $\forall z \in T@s.a \geq_{\varnothing} z$. Since $s < p$ and $a \geq_{\varnothing} p$, $a >_{\varnothing} s$. From $a >_{\varnothing} s$ it follows also that $\forall s \to n \in E \geq_{\varnothing} n$. Therefore, regardless whether $s$ has a uniform or varying branch it holds that $a \geq_{\varnothing} t$, which contradicts the assumption.

- **Case 3.** $s \to p \notin \text{cdep}(a) \wedge a \not\exists_{\varnothing} s$. We know that $s \to p \notin E$ because otherwise $s \to p$ would be a control dependence of $a$. Hence, there must be a different $q \in V$ with $q \to p \in E$, such that $p \notin T@q$ but $(\text{next}@q,p) \in D@q$ after the update of $D$ in the iteration of $q$.

As $a \geq_{\varnothing} p$, also $a \geq_{\varnothing} q$. To see why assume that $a \not\exists_{\varnothing} q$ and so $q \to p \in \text{cdep}(a)$. By the outer induction hypothesis $q$ must have a uniform branch and $T@q = \emptyset$. However, in that case $p$ was never added as a deferral target in line 10. Therefore, $a \geq_{\varnothing} q$.

Since $p = \text{next}@s$ and $s \to p \notin E$, there must be in particular such a node $q$ with $q \to p \in E$ and a path $\pi' \in q \to x \to s$ in $Gr$. Note that for every node $m \in \pi'$ it holds that next$m \in T@m$ or next$m$ is an immediate successor of $m$. By the inner induction hypothesis and $a >_{\varnothing} m$, it follows that $\forall t \in T@m.a \geq_{\varnothing} t$. Likewise, since $a >_{\varnothing} m$ also $a \geq_{\varnothing} \text{next}@m$ if next$m$ is an immediate successor of $m$. Finally, $x \in \pi'$ and next$x = s$ and so also $a \geq_{\varnothing} s$. This contradicts the assumption of the case that $a \not\exists_{\varnothing} s$. Hence, Case 3 can never occur. \hfill $\Box$

**Lemma B.6.** If $\text{uni}(k)$ with $k \in V$ then for all $b \in V$, $k \geq_{\varnothing} b \implies (\forall t \in T@b.k \geq_{\varnothing} t)$.

**Proof.** This is the inner induction hypothesis of Lemma B.5. It is thus proved by the accompanying proof of that Lemma. We will use the induction hypothesis as a standalone argument and thus rephrase it here as a corollary. \hfill $\Box$

**Lemma B.7.** If $\text{uni}(k)$ with $k \in V$ then for all $b \in V$, $(\exists t \in T@b.(k \geq_{\varnothing} t)) \implies k \geq_{\varnothing} b$

**Proof.** We will prove the claim by induction over the block index.

**Base case** The base case is given for instances where $T@b = \emptyset$, which includes the entry block of the CFG. If $T@b = \emptyset$ then $\forall t \in T@b.(k \geq_{\varnothing} t)$.

**Induction step** We prove the induction step for $b \in V$. Since $T@b \neq \emptyset$, the node $b = \text{next}@p$ for some $p \in V$ with $p < b$. When each such $p$ is processed by partial linearization, it will add new entries of the form $(b,d)$ to the deferral relation that result in entries $d \in T@b$. Note that $D = \emptyset$ initially, and these transfers by nodes $p$ with $\text{next}@p = b$ are the only way to add elements to $T@b$.

We thus distinguish the following cases for $t \in T@b$ with $k \geq_{\varnothing} t$ where $(b,t)$ was added to the deferral relation for a node $p$ with $\text{next}@p = b$.

Case 1. $\exists i.(p,i,b) \in E$

If $k \geq_{\varnothing} t$ for $t \in T@p$ then by the induction hypothesis, $k \geq_{\varnothing} p$. Further, since $p \to b \in E$, immediately $k \geq_{\varnothing} b$.

Case 2. $\exists i.(p,i,b) \in E$

In this case $b = \text{next}@p \in T@p$. By the induction hypothesis with $t \in T@p$, $k \geq_{\varnothing} p$. So, it follows from Lemma B.6 with $\text{uni}(k)$ that $\forall t \in T@p.k \geq_{\varnothing} t$ and in particular $k \geq_{\varnothing}$ next$p = b$.

\hfill $\Box$

**Lemma B.8.** If $\forall a \to b \in E.\text{uni}(a \to b)$ then $\forall b.a \to b \in E \implies a \to b \in E_f$

**Proof.** $\text{uni}(a \to b)$ implies that $a$ has a uniform branch and thus $\forall a \to b \in E.\text{uni}(a \to b)$. Since $\text{uni}(a)$ then $T@a = \emptyset$ by Lemma B.5. Because of that $a \to b \in E$ implies $a \to b \in E_f$ by the algorithm. This means that $|\{b \mid a \to b \in E_f\}| \geq |\{b \mid a \to b \in E\}|$. However, the algorithm will only reduce the degree of branches. This means that $|\{b \mid a \to b \in E_f\}| \leq |\{b \mid a \to b \in E\}|$. Thus, $\forall b.(a \to b \in E \implies a \to b \in E_f)$.\hfill $\Box$

**Lemma B.9.** if $\text{uni}(a)$ then $a \geq_{\varnothing} b \iff a \geq_{\varnothing} b$

**Proof.** We prove the claim by induction over the post dominance relation in $Gr$. The induction hypothesis is as follows with induction performed over the node $b$ with an arbitrary but fixed node $a$:

If $\text{uni}(a)$ then $a \geq_{\varnothing} b \implies a \geq_{\varnothing} b$.

In the following assume $\text{uni}(a)$. The base case is given by the roots of the post-dominator tree that is the $b \in V$, such that there is no $a$ with $a \geq_{\varnothing} b$.\hfill $\Box$
Base case.Lemma B.4 implies that $a \succeq_{\ell} b \implies a \succeq_{\ell} b$. Since $b$ is a root of the post-dominator tree, there is no other $a \in V$ with $a \succeq_{\ell} b$ but $a = b$ and so it follows that $a \succeq_{\ell} b$.

Induction step. For the induction step, we will show the contraposition $a \not\succeq_{\ell} b \implies a \not\succeq_{\ell} b$. Given that $a \not\succeq_{\ell} b$ and $b$ is processed in the outer loop, we distinguish the following cases:

Case 1. There exists $(b, i, \text{next}@b) \in E_{\ell}$ with $\text{next}@b \in T@b$. In this case, it follows directly from Lemma B.7 that $a \not\succeq_{\ell} b$ implies $a \not\succeq_{\ell} \text{next}@b$. By the induction hypothesis for $\text{next}@b$, we conclude that $a \not\succeq_{\ell} \text{next}@b$. Since $b \rightarrow \text{next}@b \in E_{\ell}$ therefore also $a \not\succeq_{\ell} b$.

Case 2. For all $(b, i, \text{next}@b) \in E_{\ell}$ it holds that $\text{next}@b \notin T@b$. In this case $\text{next}@b$ is drawn from the immediate successors of $b$ in $G$.

Sub case 2.1. $b$ has a divergent branch. Assume there was a $b \rightarrow s \in E$ with $a \not\succeq_{\ell} b$ and $a \succeq_{\ell} s$. This implies that $b \rightarrow s \in cdep(a)$. However, as $\text{unip}(a)$ the node $b$ must have a uniform branch, which contradicts the assumption. Therefore, such an edge can not exist and thus if $b$ has a divergent branch it follows from $a \not\succeq_{\ell} b$ that $V_b \rightarrow s \in E.a \not\succeq_{\ell} s$. So, if $b \rightarrow \text{next}@b \in E$ then $a \not\succeq_{\ell} \text{next}@b$. We apply the induction hypothesis to obtain $a \not\succeq_{\ell} \text{next}@b$ and finally $a \not\succeq_{\ell} b$.

Sub case 2.2. $b$ has a uniform branch. Since $a \not\succeq_{\ell} b$ there must be an edge $b \rightarrow s \in E$ such that $a \not\succeq_{\ell} s$. By assumption of Case 2, the node $s$ is also an immediate successor of $b$ in $G_{\ell}$. By the induction hypothesis $a \not\succeq_{\ell} s$. Therefore, also $a \not\succeq_{\ell} b$. \qed

B.2 Main Proof

This is the main proof of Lemma B.1.

Proof. In the following we will assume that $\text{unip}(c)$ for some $c \in V$. We will prove the two directions of the equivalence separately, that is $A \iff B \iff A$.

Direction $a \rightarrow b \in cdep(c) \implies a \rightarrow b \in cdep(c)$

By definition of control dependence, we obtain $c \succeq_{\ell} b$ and $c \not\succeq_{\ell} a$ and $a \rightarrow b \in E_{\ell}$. By Lemma B.4 and Lemma B.9, given that $c \succeq_{\ell} b$, it follows that $c \succeq_{\ell} b$ and $c \not\succeq_{\ell} a$. It remains to show that $a \rightarrow b \in E_{\ell}$. Assume this was not the case, that is $a \rightarrow b \in E_{\ell}$ and $a \rightarrow b \notin E_{\ell}$. As $a \rightarrow b \in E_{\ell}$, we get $b \in T@a$ and therefore, by Lemma B.3, $b \succeq_{\ell} a$. Since also $c \succeq_{\ell} b$ this contradicts the assumption that $c \not\succeq_{\ell} a$. Thus, $a \rightarrow b \in E_{\ell}$.

Finally, from $a \rightarrow b \in E_{\ell}$ and $c \succeq_{\ell} b$ and $c \not\succeq_{\ell} a$ it follows by definition that $a \rightarrow b \in cdep(c)$.

Direction $a \rightarrow b \in cdep(c) \implies a \rightarrow b \in cdep(c)$

Given $a \rightarrow b \in cdep(c)$ and $\text{unip}(c)$ we conclude that $\text{unip}(a \rightarrow b)$. Therefore, by Lemma B.8, $a \rightarrow b \in E_{\ell}$ because $a$ has a uniform branch and $a \rightarrow b \in E_{\ell}$. $a \rightarrow b \in cdep(c)$ also implies $c \succeq_{\ell} b$ and $c \not\succeq_{\ell} a$ by definition of control dependence. However, by Lemma B.4, $c \succeq_{\ell} b$ implies $c \succeq_{\ell} b$ and since $\text{unip}(c)$ it also follows by Lemma B.9 that $c \not\succeq_{\ell} a$ implies $c \not\succeq_{\ell} a$. In short, $a \rightarrow b \in E_{\ell}$ and $c \succeq_{\ell} b$ and $c \not\succeq_{\ell} a$ and so by definition $a \rightarrow b \in cdep(c)$.

C Preservation of Uniform Branches

Theorem C.1. Given a dominance-compact block index, partial linearization will preserve an edge $b \rightarrow y \in E$ if $\text{unip}(b)$ or there exists a block $d \in V$ with the following properties in $G$:

1. $d \succeq_{\ell} b \land d \succeq_{\ell} y$ ($d$ dominates the edge $b \rightarrow y$).
2. $\text{unip}(b \rightarrow y)$ in the dominance region $G_{\ell}$ of $d$.

In this section, we will prove Theorem C.1. We will prove that the edges that $d \in V$ dominates in the partially linearized subgraph $G_{\ell}^{d}$ are part of the whole linearized subgraph $G_{\ell}$. The proof considers two instances of partial linearization, one on $G$ and the other on $G_{\ell}$ and shows that they maintain an equivalent state with respect to the equivalence relation of Definition C.2.

We will show inductively that the equivalence relation holds when executing the two instances in lockstep for each visited note $b \in V$. This the lock step execution over the outer loop (line 3) and the inner loop (line 7) in case that $b$ ends in a uniform branch. We pad the loop of the instance on $G_{\ell}$ with empty loop iterations for blocks $b \in V \setminus V_{d}$ and edges $e \in E \setminus E_{\ell}$ such that both instances can execute in lockstep over all of $b \in V$. Note that the two instances operate on the same block index, that is $\text{Index}(b) = \text{Index}_{\ell}(b)$ for $b \in V_{d}$.

Finally, the equivalence relation implies that all edges in $G_{\ell}^{d}$ that $d$ dominates are indeed embedded in $G_{\ell}$. By extension if an edge $a \rightarrow b \in E$ with $d \succeq_{\ell} b$ is uniform in $G_{\ell}$ for any node $d \in V$ then it will be preserved in $G_{\ell}^{d}$ and thus also in the whole partially linearized $G_{\ell}$.

Definition C.2. The instances of the partial linearization algorithm on $G$ and on $G_{\ell}$ are in an equivalent state at the outer loop iteration for block $b$, if $D_{\ell}^{b} \sim D@b$ and $E_{\ell}^{b} \sim E_{\ell}@b$ where these are defined as:

$$\forall x, d \succeq_{\ell} y. \left( (x, y) \in D_{\ell}^{b} \iff (x, y) \in D@b \right)$$

$$\forall x, d \succeq_{\ell} y. \left( (x, y) \in E_{\ell}^{b} \iff (x, y) \in E_{\ell}@b \right)$$

C.1 Main Proof

Theorem C.3. Partial linearization maintains the equivalence relation of Definition C.2.

Proof. We will prove this by induction over the two instances of the algorithm. The induction hypothesis states that the equivalence relation of Definition C.2 holds before a new
We need to show that the equivalence relation still holds after the outer loop iteration for a block \( b \in V \).

**Base case (first block)**  The equivalence relation holds before the first outer loop iteration because up to line 3 \( D^d = D = \emptyset \) and \( E^d_T = E_T = \emptyset \).

**Induction step (case \( d^0 \) next@b)**  \( D^d@b \sim D@b \) Assume there was a \( (\text{next@b}, y) \in D@b \) with \( d^0 \) \( y \) after the outer loop iteration for \( b \). Then \( \text{next@b} < y \) and further \( \text{next@b} < d \) because the block index is dominance compact. There must be an edge \( p \to y \in E \) with \( p \leq b \) \( \text{next@b} \leq d \) because either \( p = b \) or \( p \) must have been processed before \( b \) to add \( y \) as a deferral target. However, if \( p < d \) then \( d^0 \) \( p \) and also \( d^0 \) \( y \), which contradicts the assumption.

\[ E^d_T @b \sim E_T @b : \text{The } \forall \text{-quantifier in the definition of } E^d_T @b \sim E_T @b \text{ does not quantify over edges } b \to \text{next@b} \in E_T @b \text{ with } d^0 \text{ next@b}. \]

These are the only kind of edges added to \( E_T \) and \( E^d_T \) in this case.

**Induction step (case \( d^>0 \) next@b)**  We first show that \( d^\geq_0 b \). The new branch target next@b either originates from the direct successors of \( b \) or from \( T@b \). So, there must be an edge \( p \to \text{next@b} \in E \) with \( p \leq b \). Since \( d^>0 \) \( \text{next@b} \) also \( d^\geq_0 p \) and \( d \leq p \). As \( b < \text{next@b} \) either \( d^\geq_0 b \) or \( b < d \). However, in case that \( b < d \) then \( b < p \) and so \( p \) has not been processed yet, which contradicts the existence of \( p \to \text{next@b} \in E \).

We now turn to the induction step. Note that the node \( b \) has the same set of successor edges in both \( G^d \) and \( G \) by the definition of \( G^d \) (Theorem 4.3). Further, \( D \sim D^d \) and \( E_T \sim E^d_T \) before line 7 for a uniform branch or line 13 for a divergent branch. Therefore, we only need to show that \( D@b = D^d@b \) for each step. Then it follows that \( D@b = D^d@b \) and \( E_T@b = E^d_T@b \) after the step.

**Case 1. Inner loop step for uniform branch in \( b \).**

Let \( (b, i, s) \in E \) be the edge in \( G^d \) and \( G \) processed by the inner loop. Because the inner loop executes in lock stepping there is no need to show that \( \text{next@b} = \text{next}^d@b \) after line 8.

Consider the case that \( \text{next@b} \in T@b \). Then, because \( d^>0 \) \( \text{next@b} \) and \( D@b \sim D^d@b \), also \( \text{next@b} \in T^d@b \). There could be a \( t \in T@b \) with \( d^>0 \) \( t \) and \( t < \text{next@b} \) since \( d^>0 \) \( b \) and \( d^>0 \) \( \text{next@b} \) and so \( t < b \), which contradicts \( t > b \). Hence, next@b = next^d@b.

**Case 2.** \( b \) has a divergent branch.

We need to show that \( \text{next@b} = \text{next}^d@b \) after line 14 where next \( \leftarrow \) min(\( T \cup S \)).

In case that \( \text{next@b} \in T@b \) there can not be a \( t \in T@b \) with \( t < \text{next@b} \) for the same reason as in the uniform case. Note that \( S@b = S^d@b \) because \( d^\geq_0 b \) and so \( \text{min}(S@b) = \text{min}(S^d@b) \). Therefore, next@b = next^d@b.

It remains to show that line 18 does not affect the equivalence relation. First, note that the expression \( D \leftarrow D_1 \{ (b, s) \mid (b, s) \in D \} \) does not add new pairs to either \( D \) or \( D^d \). Finally, if before the line there was an edge \( (b, z) \in D@b \) and \( (b, z) \in D^d@b \) with \( d^>0 z \), it will be removed from both \( D@b \) and \( D^d@b \).

Therefore, both instances are in equivalent state after an outer loop iteration on \( b \in V \).
M. S. Hecht and J. D. Ullman. 1974. Characterizations of Reducible

Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Chris Lattner and Vikram Adve. 2004. LLVM: A compilation frame-


Christian Wimmer and Hanspeter Mössenböck. 2005. Optimized Interval Splitting in a Linear Scan Register Allocators. In Proceedings...

https://doi.org/10.1145/1064979.1064998