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:
Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3192366.3192413

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.

```
1  int search(Node * nodes, float * Q, int i) {
2    int stack[512]; stack[0] = 0;
3    int top = 1;
4
5    float elem = Q[i];
6    int result = -1;
7
8    while (top > 0) {
9      int next = stack[--top];
10     float label = nodes[next].data;
11     int right = nodes[next].right;
12     int left = nodes[next].left;
13
14     if (label == elem) {
15       result = next; break;
16     }
17     if (any(elem < label) && left > 0)
18       stack[top++] = left;
19     if (any(label < elem) && right > 0)
20       stack[top++] = right;
21   }
22   return result;
23 }
```

Figure 1. Data-parallel binary tree search.

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[1]. 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 $any(v)$ evaluates to true for *all* SIMD lanes, if v evaluates to true for *any* SIMD lane. Otherwise, $any(v)$ 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 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 (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 $entry \in V$ 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 then 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 π 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 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 $a \succeq^D b$, iff 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 $a \succeq^{PD} b$, 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 $k \succeq^{PD} b$ and $k \not\succeq^{PD} a$. 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].

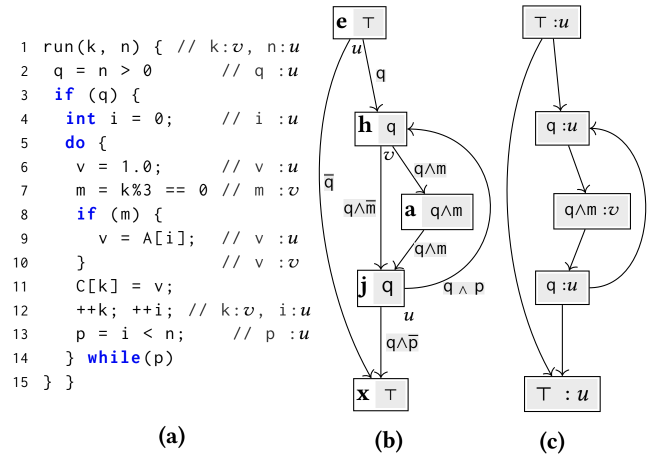
2.2 Vectorization of Data-Parallel CFGs

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



```
 1 run(k, n) { // k:v, n:u
 2   q = n > 0        // q :u
 3   if (q) {
 4     int i = 0;     // i :u
 5     do {
 6       v = 1.0;      // v :u
 7       m = k%3 == 0 // m :v
 8       if (m) {
 9         v = A[i];   // v :u
10       }             // v :v
11       C[k] = v;
12       ++k; ++i; // k:v, i:u
13       p = i < n;     // p :u
14     } while(p)
15 } }
```

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 $q \wedge m$ 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 $b \in V$ and all edges $a \longrightarrow b \in E$. The predicate for an edge $a \longrightarrow b$ 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 $a \in V$. We call an edge $a \longrightarrow b \in E$ uniform, written $uni(a \longrightarrow b)$, iff $uni(a)$ and block b ends in a uniform branch. Iff the constituents of a block predicate are all uniform, then the predicate of the block itself is uniform, i.e. $uni(a) \iff uni(cdep(a))$.

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 $Index : Blocks \longrightarrow \{0, .., n-1\}$ 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 $B \subset Blocks$, iff all blocks in the range of

$$[\min\{Index(b) \mid b \in B\}, \max\{Index(b) \mid b \in B\}]$$

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.

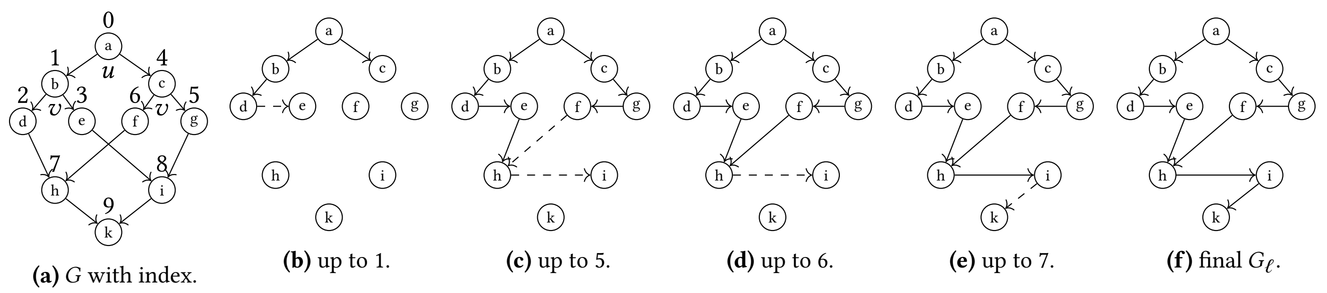


Figure 3. Walkthrough of partial linearization. (a) source CFG G [21] with divergent branches. (b)-(f) partially linearized CFG G_ℓ after the specified iteration (block index number). Deferral edges are shown as dashed arrows.

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 $G_\ell = (V, E_\ell, entry)$ that constitutes a partially if-converted version of the original graph $G = (V, E, entry)$. Coming back to the example, the initial graph is shown in Figure 3a and the final graph G_ℓ 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_ℓ, 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_ℓ. However, if a path in G_ℓ 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_ℓ it has to make sure that all other successors of b in the original graph will post-dominate b in G_ℓ 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_ℓ (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,

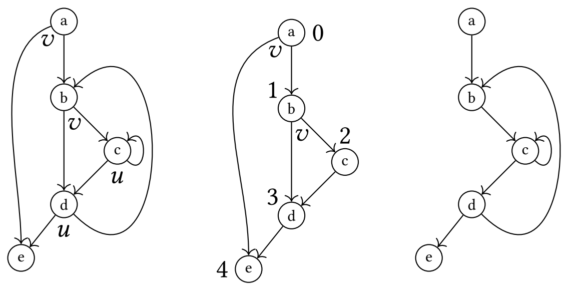


Figure 4. Handling of loops in partial linearization. Left: G with nested uniform loops. Center: backedges are removed, shown with loop compact block index. Right: G_ℓ with re-inserted backedges.

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_ℓ. 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_ℓ 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

```
   Input: CFG $G = (V, E, entry)$
   Input: Block index of $G$ (see Section 3.1)
   Output: Partially linearized CFG $G_\ell = (V, E_\ell, entry)$
 1 // $P \leftarrow \emptyset$
 2 $D \leftarrow \emptyset$
 3 foreach $b$ in Index do
 4     // $F \leftarrow \{v \mid \exists u.(u, v) \in D\}$
 5     $T \leftarrow \{s \mid (b, s) \in D\}$
 6     if $b$ ends in a uniform branch then
 7         foreach $(b, i, s) \in E$ do
 8             $next \leftarrow \min(T \cup \{s\})$
 9             $E_\ell \leftarrow E_\ell \cup \{(b, i, next)\}$
10             $D \leftarrow D \cup \{(next, t) \mid t \in (T \cup \{s\}) \setminus \{next\}\}$
11         end
12     else
13         $S \leftarrow \{s \mid \exists i.(b, i, s) \in E\}$
14         $next \leftarrow \min(T \cup S)$
15         $E_\ell \leftarrow E_\ell \cup \{(b, 0, next)\}$
16         $D \leftarrow D \cup \{(next, t) \mid t \in (T \cup S) \setminus \{next\}\}$
17     end
18     $D \leftarrow D \setminus \{(b, s) \mid (b, s) \in D\}$
19     // $P \leftarrow P \cup \{b\}$
20 end
```

Figure 5. Partial linearization algorithm. Lines 1,4,19 are abbreviations used in the proofs.

inside the blocks by predication or masking. Note that predication is orthogonal to producing the CFG itself and we will assume a correct predication of the code in the following. On this assumption, the transformed program is correct if each path of the original CFG appears as a sub-path in the partially linearized one. In the remainder of this section we will prove that this is indeed the case.

We will first show that every path in the scalar CFG is part of a path in the partially linearized CFG. The proof is carried out by induction and uses the following invariant of the outer loop.

Lemma 3.1. *For each node v that has a predecessor p in P, it holds for v that there is either an edge $(p, b) \in E_\ell$ or there is another node p' for which there is a path from p to p' in $E_\ell \cup D$ and $(p', b) \in D$.*

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_ℓ. If *next* is picked to be s, then the edge (b, s) is added to G_ℓ (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 $E_\ell \cup D$) 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 π in $E_\ell \cup D$ 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 in b are "re-originated" to *next*. because the edge $(b, next)$ is added to E_ℓ, the to-be-removed edge (b, t) can be replaced by the two-edge path $b, next, t$ in π. Hence the property is preserved for all other nodes unequal to b. □

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

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 $\pi \in entry \longrightarrow^* p$ in G where p is a predecessor of b. By the induction hypothesis, there is also a path π' in G_ℓ that contains π as a subpath. Consider the extension $\pi \circ (p \longrightarrow b)$ of π to b. By Lemma 3.1, there is either an edge $(p, b) \in E_\ell$ or a path $p \longrightarrow^* b$ in $E_\ell \cup D$. □

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

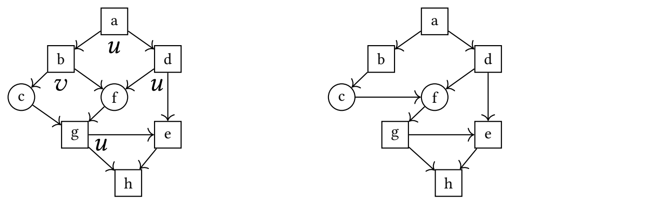


Figure 6. Left: G, the source CFG with uniform predicates in rectangle nodes, right: G_ℓ, the partially linearized version of G. The code generator can ignore all predicates, except for those in c and f

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 $uni(b)$, i.e. the predicate of a block $b \in V$ is uniform, then execution will reach block b in G_ℓ iff the predicate of b is true.*

The proof makes use of Lemma 4.2, which states that if $uni(k)$ then the control dependences of k are preserved in G_ℓ. 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 $uni(k)$ then $cdep(k) = cdep_\ell(k)$ where $cdep_\ell$ is the control dependence in G_ℓ.*

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_ℓ. This follows from the correctness of partial linearization that if π is a path in G with $k \in \pi$ then π is embedded in a path π' in G_ℓ with $k \in \pi'$.

It remains to show that if execution reaches the block k in G_ℓ 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 $cdep(k) = \emptyset$ then k is always executed in G. Since every path in G is embedded in a path in G_ℓ, the block k is also always executed in G_ℓ. Note that $cdep(entry) = \emptyset$ for $entry$, the first block in the block index.

Induction step: Assume $uni(k)$ for some $k \in V$. We need to show that if k is executed in G_ℓ then k is also executed in G.

Let $\pi' \in entry \longrightarrow^* k$ be an arbitrary prefix path to k in G_ℓ. Then, there is an edge $a \longrightarrow b \in cdep_\ell(k)$ such that $\pi' \in entry \longrightarrow^* a \longrightarrow b \longrightarrow^* k$.

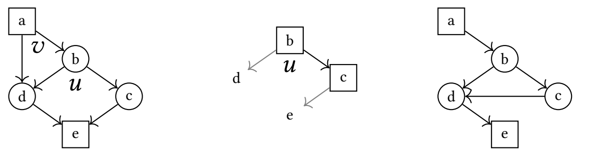


Figure 7. Left: source CFG; center: G^b the dominance subgraph of b; right: preserved uniform branch in b after partial linearization.

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

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

However, $a \longrightarrow b \in cdep(k)$ implies that $k \geq^{PD} b$ and thus any complete path in G that contains b will eventually pass through k. Hence, if $uni(k)$ and k is executed in G_ℓ 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 $a \longrightarrow b$, 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 $G = (V, E, entry)$ that $d \in V$ dominates:
$E^d = \{x \longrightarrow y \in E \mid d \geq^D x\}$
$V^d = \{x \in V \mid d \geq^D x \vee (\exists y. y \longrightarrow x \in E^d)\}$

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^d of b in the center. The block b dominates c and so the edge $b \longrightarrow c$ will be preserved. Generally, as stated by Theorem 4.5, if an edge $a \longrightarrow b$ is uniform relative to

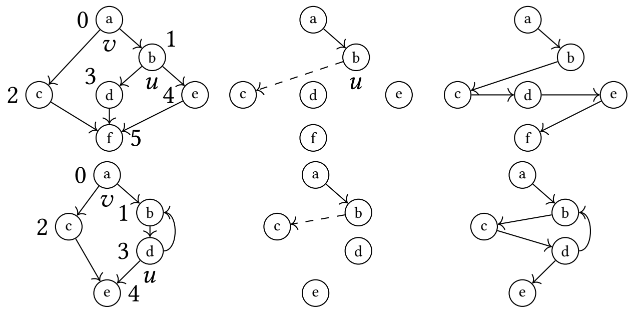


Figure 8. Top: Effect of non dominance-compact block index. Bottom: Effect of non loop-compact block index. Left: original CFGs G with (non compact) block index, Center: processed up to 1, Right: G_ℓ with defect.

a node d and d dominates the edge then the edge will be preserved.

Theorem 4.5. *Given a dominance-compact block index, partial linearization will preserve an edge $b \longrightarrow y \in E$ if uni(b) or there exists a block $d \in V$ with the following properties in G:*

1. $d \succeq^D b \wedge d \succ^D y$ *(d dominates the edge $b \longrightarrow y$).*
2. *uni($b \longrightarrow y$) 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 $b \longrightarrow d$ and $b \longrightarrow e$. 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.

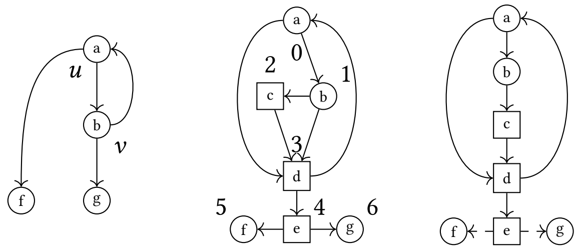


Figure 10. Divergent loop transform on the Mandelbrot example. Left: scalar CFG. Center: after divergent loop transform with block index. Right: partial linearization up to 3.

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

```
1 for (i = 0; i < Limit; ++i) {
2   z = z * z + c;
3   if (hypot(__real__ z, __imag__ z) >= ESCAPE)
4     break;
5 }
```

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 9 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 $\phi_{liveMask}$ node in the loop header to track the live threads in the loop. For each exit to a block x, another mask $\phi_{xExitMsk}$ 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 $\phi_{fExitMsk}$ and $\phi_{gExitMsk}$. 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

```
$\phi_{liveUpd} \leftarrow [\phi_{liveMask}, b], [0, c], [0, a]$
$\phi_{fExitUpd} \leftarrow [\phi_{liveMsk}, b], [\phi_{fExitMsk}, c], [\phi_{fExitMsk}, a]$
$\phi_{gExitUpd} \leftarrow [\phi_{gExitMsk}, b], [\phi_{cMask}, c], [\phi_{gExitMsk}, a]$
$\phi_{iOut} \leftarrow [\phi_{iTrack}, b], [\phi_i, c], [\phi_i, a]$
br any($\phi_{liveUpd}$) a e
```

Figure 11. pure latch block (d) with mask update ϕ.

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 ϕ nodes in the pure latch block. The node $\phi_{liveUpd}$ 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 $\phi_{fExitUpd}$, $\phi_{gExitUpd}$ 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 ϕ 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 ϕ-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*.

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

Figure 12. Structure of hot loop in SPEC2017 644.nab_s with branch probabilities (`if`-case taken).

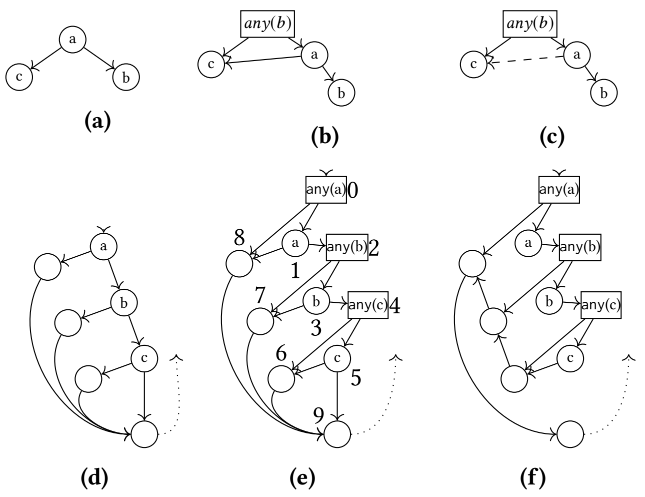


Figure 13. (a) divergent branch in a, (b) BOSCC gadget to skip b, (c) deferral relation at node a. (d) Excerpt CFG from hot loop in nab (Listing 12, Line 11 till end). (e) With three nested BOSCC gadgets. (f) After partial linearization.

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 *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 *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 a 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].

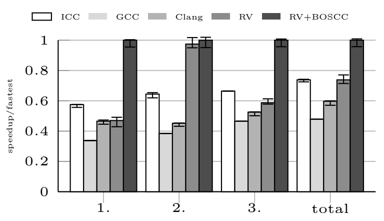
[2] https://github.com/cdl-saarland/rv

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.

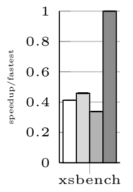
Benchmarks We adopted the Vantage Point, Nearest Neighbor, Point Correlation, k-means clustering and Barnes-Hut data analytics kernels and data sets from the existing *Lonestar* [24] and *Treelogy* [18] benchmark suites and added two new benchmarks: multi-radius point correlation (mpc) and binary tree (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.
- **Nearest Neighbor (nn)** Nearest Neighbor on a kd tree.
  random: 1000,000 random points (diameter 141.421).
  geocity: 2,673,765 city coordinates (diameter 385.32).
  covtype: 581,012 data points with nine integer features from a tree coverage data set [7] (diameter 10246.1).
- **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 with in 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 quarry. This is the inner-most loop of the *XSBench* benchmark [41]. *random:* 4,194,304 elements.

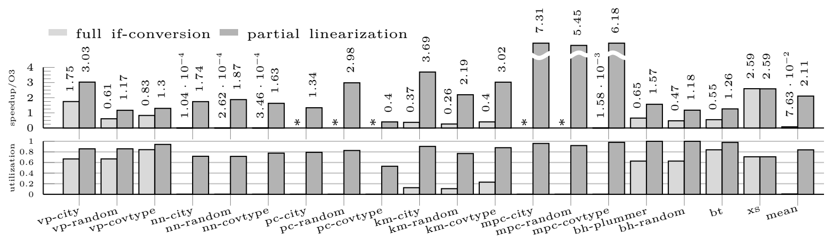
Multi-Radius Point Correlation For the *bh*, *nn*, *pc*, *vp* and *km* benchmarks, the query coordinate is always *varying* while all other parameters to the query are uniform. It has been noted [14] that some machine learning applications benefit from a SIMD version of Point Correlation that takes a 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.



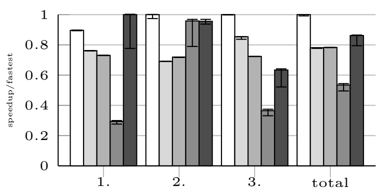
(a) `644.nab_s`



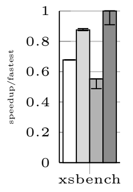
(b) XSBench



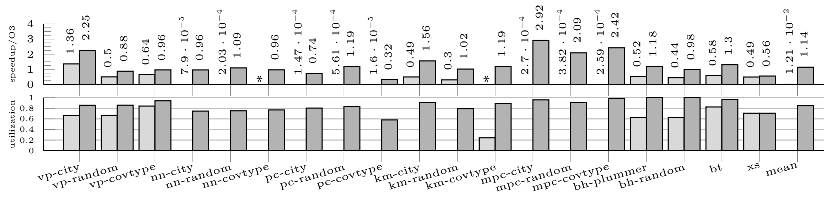
(c) Treelogy (AVX512 - Skylake)



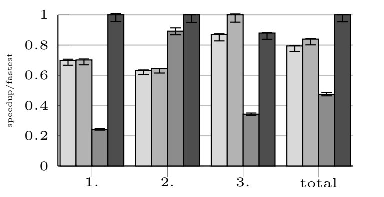
(d) `644.nab_s`



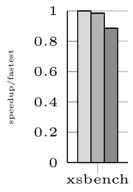
(e) XSBench



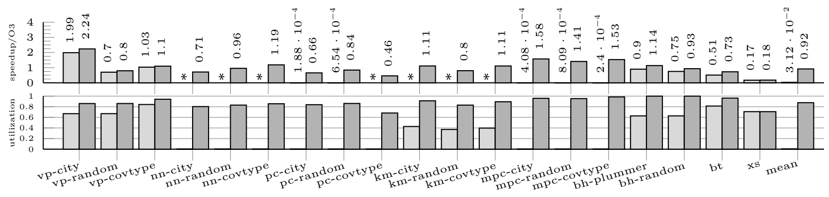
(f) Treelogy (AVX2 - Haswell)



(g) `644.nab_s`



(h) XSBench



(i) Treelogy (Adv. SIMD - Cortex-A53)

Figure 14. Running time results.

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 4096 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^{20} 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.

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.

Name	**Branch**			**Loop**		**Exit**		**L/S Masks**		
	div	lost	pres	uni	div	uni	div	true	uni	var
bh	0	0	3	2	0	0	0	14	6	0
bt	2	0	4	0	1	1	1	3	0	6
km	8	0	8	3	2	2	2	14	35	6
mpc	4	0	13	6	2	2	2	6	52	6
nn	8	0	8	3	2	2	2	14	35	6
pc	8	0	6	3	2	2	2	14	32	6
vp	1	0	3	2	0	0	0	16	3	1
xs	1	0	1	0	1	0	1	0	0	1
nab-1/vec	7	0	3	1	0	0	0	6	4	11
nab-1/bsc	7	0	6	1	0	0	0	6	4	11
nab-2/vec	2	0	3	1	0	0	0	31	10	20
nab-2/bsc	2	0	3	1	0	0	0	31	10	20
nab-3/vec	7	0	2	1	0	0	0	37	14	22
nab-3/bsc	7	0	5	1	0	0	0	37	14	22
xsbench	1	0	1	2	1	0	1	46	0	2

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 partial 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. 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.

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 *reftrain* 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 (*aminos* 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: XSBench

XSBench 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 XSBench with the `nuclide` 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%

(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 amendable 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 loops. 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 $cdepB(k) = \{a \in V \mid \exists b.a \to b \in cdep(k)\}$

We write $\succeq_\ell^{PD}$ and $cdep_\ell$ to refer to post dominance and control dependence on the partially linearized graph G_ℓ.

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_ℓ. 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 $uni(k)$ then $cdep(k) = cdep_\ell(k)$ where $cdep_\ell$ is the control dependence in G_ℓ.*

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 $\forall s \in S@b.x \leq s$.
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. $c \in T@b \implies c \succ_\ell^{PD} b$

Proof. Given that $c \in T@b$, consider every complete path $\pi \in b\downarrow$ in G_ℓ. Since π is complete it ends in some $x \in V$ where x is a block without successors in G_ℓ. 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_ℓ. As this reasoning applies to any complete path π from b in G_ℓ, the node c is element of any such path π. Thus, by definition of post dominance, $c \succ_\ell^{PD} b$. □

Lemma B.4. $a \succeq^{PD} x \implies a \succeq_\ell^{PD} x$

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 $a \succ^{PD} x$. For every successor p with $x \to p$ in E it holds that $a \succeq^{PD} p$. By the induction hypothesis therefore $a \succeq_\ell^{PD} p$. For every edge $(x, i, next@x) \in E_\ell$ there are two cases: Either immediately $next@x = p$ or it holds that $next@x \neq p$. In the latter case $(next@x, p) \in D@x$ after the update to D and so $p \succ_\ell^{PD} next@x$ by Lemma B.3 with $a \succeq_\ell^{PD} p$. Therefore, in general $a \succ_\ell^{PD} x$. □

Lemma B.5. $uni(a) \implies T@a = \emptyset$

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 $uni(a) \implies T = \emptyset$.

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 $uni(a)$ it holds that $\forall d \in cdepB(a).T@d = \emptyset$. This is because $uni(a)$ implies $uni(cdep(a))$. 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 $a \succeq^{PD} p \implies (\forall t \in T@p.a \succeq^{PD} t)$. 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 $a \succeq^{PD} p$. 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 $s \to p \in E$ during which processing the pair $(next@s, p)$ was inserted into the deferral relation. Then, $a \not\succeq^{PD} s$ because p is the minimum node with $a \succeq^{PD} p$ and hence $s \to p \in cdep(a)$.

With $uni(a)$ it follows that s has a uniform branch and the outer induction hypothesis implies that $T@s = \emptyset$. Therefore, always $(next@s, p) \notin D@s$ *after* line 10, for any such $s \rightarrow p \in E$. This contradicts $p \in T@x$ for any $x \in V$.

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

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

We distinguish three cases for s:

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

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

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

As $a \succeq^{PD} p$, also $a \succeq^{PD} q$. To see why assume that $a \not\succeq^{PD} q$ and so $q \rightarrow p \in cdep(a)$. 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, $a \succeq^{PD} q$.

Since $p = next@s$ and $s \rightarrow p \notin E$, there must be in particular such a node q with $q \rightarrow p \in E$ and a path $\pi' \in q \rightarrow^* x \rightarrow s$ in G_ℓ. 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 $a \succ^{PD} m$, it follows that $\forall t \in T@m.a \succeq^{PD} t$. Likewise, since $a \succ^{PD} m$ also $a \succeq^{PD} next@m$ if $next@m$ is an immediate successor of m. Finally, $x \in \pi'$ and $next@x = s$ and so also $a \succeq^{PD} s$. This contradicts the assumption of the case that $a \not\succeq^{PD} s$. Hence, *Case 3* can never occur. □

Lemma B.6. *if $uni(k)$ with $k \in V$*
then for all $b \in V$, $k \succeq^{PD} b \implies (\forall t \in T@b.k \succeq^{PD} t)$.

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. □

Lemma B.7. *If $uni(k)$ with $k \in V$*
then for all $b \in V$, $[\exists t \in T@b. (k \succeq^{PD} t)] \implies k \succeq^{PD} b$

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 $\forall t \in T@b.(k \not\succeq^{PD} t)$.

Induction step We prove the induction step for $b \in V$. Since $T@b \neq \emptyset$, the node $b = next@p$ 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 $next@p = b$ are the only way to add elements to $T@b$.

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

Case 1. $\exists i.(p, i, b) \in E$
If $k \succeq^{PD} t$ for $t \in T@p$ then by the induction hypothesis, $k \succeq^{PD} p$. Further, since $p \rightarrow b \in E$, immediately $k \succeq^{PD} b$.

Case 2. $\nexists i.(p, i, b) \in E$
In this case $b = next@p \in T@p$. By the induction hypothesis with $t \in T@p$, $k \succeq^{PD} p$. So, it follows from Lemma B.6 with $uni(k)$ that $\forall t \in T@p.k \succeq^{PD} t$ and in particular $k \succeq^{PD} next@p = b$.

□

Lemma B.8. *if $\forall a \rightarrow b \in E.uni(a \rightarrow b)$ then*
$\forall b.a \rightarrow b \in E \iff a \rightarrow b \in E_\ell$

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

Lemma B.9. *if $uni(a)$ then $\left[a \succeq^{PD} b \impliedby a \succeq^{PD}_\ell b\right]$*

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

$$\text{If } uni(a) \text{ then } a \succeq^{PD}_\ell b \implies a \succeq^{PD} b.$$

In the following assume $uni(a)$. 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 $a \succ^{PD}_\ell b$.

Base case Lemma B.4 implies that $a \succeq^{PD} b \implies a \succeq^{PD}_{\ell} b$. Since b is a root of the post-dominator tree, there is no other $a \in V$ with $a \succeq^{PD}_{\ell} b$ but $a = b$ and so it follows that $a \succeq^{PD} b$.

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

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

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

Sub case 2.1. b has a divergent branch.
Assume there was a $b \to s \in E$ with $a \not\succeq^{PD} b$ and $a \succeq^{PD} s$. This implies that $b \to s \in cdep(a)$. However, as $uni(a)$ 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 $a \not\succeq^{PD} b$ that $\forall b \to s \in E. a \not\succeq^{PD} s$. So, if $b \to next@b \in E$ then $a \not\succeq^{PD} next@b$. We apply the induction hypothesis to obtain $a \not\succeq^{PD}_{\ell} next@b$ and finally $a \not\succeq^{PD}_{\ell} b$.

Sub case 2.2. b has a uniform branch.
Since $a \not\succeq^{PD} b$ there must be an edge $b \to s \in E$ such that $a \not\succeq^{PD} s$. By assumption of *Case 2*, the node s is also an immediate successor of b in G_ℓ. By the induction hypothesis $a \not\succeq^{PD}_{\ell} s$. Therefore, also $a \not\succeq^{PD}_{\ell} b$. □

B.2 Main Proof

This is the main proof of Lemma B.1.

Proof. In the following we will assume that $uni(c)$ for some $c \in V$. We will prove the two directions of the equivalence separately, that is $A \implies B$ and $B \implies A$.

Direction: $a \to b \in cdep_\ell(c) \implies a \to b \in cdep(c)$
By definition of control dependence, we obtain $c \succeq^{PD}_{\ell} b$ and $c \not\succeq^{PD}_{\ell} a$ and $a \to b \in E_\ell$. By Lemma B.4 and Lemma B.9, given that $uni(c)$, it follows that $c \succeq^{PD} b$ and $c \not\succeq^{PD} a$. It remains to show that $a \to b \in E$. Assume this was not the case, that is $a \to b \in E_\ell$ and $a \to b \notin E$. As $a \to b \in E_\ell$, we get $b \in T@a$ and therefore, by Lemma B.3, $b \succeq^{PD}_{\ell} a$. Since also $c \succeq^{PD}_{\ell} b$ this contradicts the assumption that $c \not\succeq^{PD}_{\ell} a$. Thus, $a \to b \in E$.

Finally, from $a \to b \in E$ and $c \succeq^{PD} b$ and $c \not\succeq^{PD} a$ it follows by definition that $a \to b \in cdep(c)$.

Direction: $a \to b \in cdep_\ell(c) \impliedby a \to b \in cdep(c)$
Given $a \to b \in cdep(c)$ and $uni(c)$ we conclude that $uni(a \to b)$. Therefore, by Lemma B.8, $a \to b \in E_\ell$ because a has a uniform branch and $a \to b \in E$. $a \to b \in cdep(c)$ also implies $c \succeq^{PD} b$ and $c \not\succeq^{PD} a$ by definition of control dependence. However, by Lemma B.4, $c \succeq^{PD} b$ implies $c \succeq^{PD}_{\ell} b$ and since $uni(c)$ it also follows by Lemma B.9 that $c \not\succeq^{PD} a$ implies $c \not\succeq^{PD}_{\ell} a$. In short, $a \to b \in E_\ell$ and $c \succeq^{PD}_{\ell} b$ and $c \not\succeq^{PD}_{\ell} a$ and so by definition $a \to b \in cdep_\ell(c)$. □

C Preservation of Uniform Branches

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

1. *$d \succeq^{D} b \wedge d \succ^{D} y$ (d dominates the edge $b \to y$).*
2. *$uni(b \to y)$ in the dominance region G^d 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^d_ℓ are part of the whole linearized subgraph G_ℓ. The proof considers two instances of partial linearization, one on G and the other on G^d 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 lock step 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^d with empty loop iterations for blocks $b \in V \setminus V^d$ and edges $e \in E \setminus E^d$ 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 $Index(b) = Index^d(b)$ for $b \in V^d$.

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

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

$D^d@b \sim D@b$ iff

$$\forall x, d \succ^{D} y. \left[(x, y) \in D^d@b \iff (x, y) \in D@b\right]$$

$E^d_\ell@b \sim E_\ell@b$ iff

$$\forall d \succeq^{D} x \wedge d \succ^{D} y. \left[x \to y \in E^d_\ell@b \iff x \to y \in E_\ell@b\right]$$

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 outer loop iteration for a block $b \in V$ in both G and G_ℓ. 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 $E^d_\ell = E_\ell = \emptyset$.

Induction step (case $d \not\succ^D next@b$) $\underline{D^d@b \sim D@b}$ Assume there was a $(next@b, y) \in D@b$ with $d \succ^D y$ *after* the outer loop iteration for b. Then $next@b < y$ and further $next@b < d$ because the block index is dominance compact. There must be an edge $p \longrightarrow y \in E$ with $p \leq b < next@b \leq d < y$ because either $p = b$ or p must have been processed before b to add y as a deferral target. However, if $p < d$ then $d \not\succeq^D p$ and also $d \not\succ^D y$, which contradicts the assumption.

$\underline{E^d_\ell@b \sim E_\ell@b}$: The $\forall$-quantifier in the definition of $E^d_\ell@b \sim E_\ell@b$ does not quantify over edges $b \longrightarrow next@b \in E_\ell@b$ with $d \not\succeq^D next@b$. These are the only kind of edges added to E_ℓ and E^d_ℓ in this case.

Induction step (case $d \succ^D next@b$) We first show that $d \succeq^D b$. The new branch target $next@b$ either originates from the direct successors of b or from $T@b$. So, there must be an edge $p \longrightarrow next@b \in E$ with $p \leq b$. Since $d \succ^D next@b$ also $d \succeq^D p$ and $d \leq p$. As $b < next@b$ either $d \succeq^D b$ 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 $p \longrightarrow next@b \in E$.

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 $E_\ell \sim E^d_\ell$ *before* line 7 for a uniform branch or line 13 for a divergent branch. Therefore, we only need to show that $next@b = next^d@b$ for each step. It then follows that $D@b \sim D^d@b$ and $E_\ell@b \sim E^d_\ell@b$ *after* the step.

<u>Case 1. Inner loop step for uniform branch in b.</u>
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 step $s@b = s^d@b$. We need to show that $next@b = next^d@b$ after line 8.

Consider the case that $next@b \in T@b$. Then, because $d \succ^D next@b$ and $D@b \sim D^d@b$, also $next@b \in T^d@b$. There could not be a $t \in T@b$ with $d \not\succ^D t$ and $t < next@b$ since $d \succeq^D b$ and $d \succ^D next@b$ and so $t < b$, which contradicts $t > b$. Hence, $next@b = next^d@b$.

<u>Case 2. b has a divergent branch.</u>
We need to show that $next@b = next^d@b$ after line 14 where $next \longleftarrow \min(T \cup S)$.

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

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

Therefore, both instances are in equivalent state *after* an outer loop iteration on $b \in V$. □

References

[1] Timo Aila and Samuli Laine. 2009. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009 (HPG '09)*. ACM, New York, NY, USA, 145–149. https://doi.org/10.1145/1572769.1572792

[2] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '83)*. ACM, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[3] Jayvant Anantpur and Govindarajan R. 2014. *Taming Control Divergence in GPUs through Control Flow Linearization.* Springer Berlin Heidelberg, Berlin, Heidelberg, 133–153. https://doi.org/10.1007/978-3-642-54807-9_8

[4] Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, and Vinod Grover. 2013. Convergence and Scalarization for Data-parallel Architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494995

[5] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-vectorization for Irregular Loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 697–710. https://doi.org/10.1145/2908080.2908111

[6] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2015. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Trans. Archit. Code Optim.* 11, 4, Article 66 (Jan. 2015), 25 pages. https://doi.org/10.1145/2693261

[7] J A Blackard and D J Dean. 1999. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture* vol.24 (1999), 131–151.

[8] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence analysis and optimizations. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on.* IEEE, 320–329.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[10] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. 2011. SIMD Re-convergence at Thread Frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 477–488. https://doi.org/10.1145/2155620.2155676

[11] Jeanne Ferrante and Mary Mace. 1985. On Linearizing Parallel Code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '85)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/318593.318636

[12] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[13] Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General Transformations for GPU Execution of Tree Traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 10, 12 pages. https://doi.org/10.1145/2503210.2503223

[14] Alexander G Gray and Andrew W Moore. 2001. N-body'problems in statistical learning. In *Advances in neural information processing systems*. 521–527.

[15] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. PACXXv2 + RV: An LLVM-based Portable High-Performance Programming Model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC'17)*. ACM, New York, NY, USA. https://doi.org/10.1145/3148173.3148185

[16] Paul Havlak. 1994. *Construction of thinned gated single-assignment form*. Springer Berlin Heidelberg, Berlin, Heidelberg, 477–499. https://doi.org/10.1007/3-540-57659-2_28

[17] M. S. Hecht and J. D. Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (July 1974), 367–375. https://doi.org/10.1145/321832.321835

[18] N. Hegde, J. Liu, and M. Kulkarni. 2016. Treelogy: a benchmark suite for tree traversal applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–2. https://doi.org/10.1109/IISWC.2016.7581286

[19] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. 2017. Fast Segmented Sort on GPUs. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/3079079.3079105

[20] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. 2013. Automatic Vectorization of Tree Traversals. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 363–374. http://dl.acm.org/citation.cfm?id=2523721.2523770

[21] Ralf Karrenberg. 2015. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer Vieweg.

[22] Ralf Karrenberg and Sebastian Hack. 2011. Whole-function Vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 141–150. http://dl.acm.org/citation.cfm?id=2190025.2190061

[23] Ralf Karrenberg and Sebastian Hack. 2012. Improving Performance of OpenCL on CPUs. In *Compiler Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[24] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*. http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[25] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

[26] Marco Lattuada and Fabrizio Ferrandi. 2017. Exploiting vectorization in high level synthesis of nested irregular loops. *Journal of Systems Architecture* 75 (2017), 1 – 14. https://doi.org/10.1016/j.sysarc.2017.03.001

[27] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler, and Krste Asanović. 2014. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 101–113. https://doi.org/10.1109/MICRO.2014.48

[28] Joseph CH Park and Mike Schlansker. 1991. *On predicated execution*. Hewlett-Packard Laboratories Palo Alto, California.

[29] M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13. https://doi.org/10.1109/InPar.2012.6339601

[30] Bin Ren, Youngjoon Jo, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2015. Efficient Execution of Recursive Programs on Commodity Vector Hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 509–520. https://doi.org/10.1145/2737924.2738004

[31] Bin Ren, Sriram Krishnamoorthy, Kunal Agrawal, and Milind Kulkarni. 2017. Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/3018743.3018763

[32] Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD Parallelization of Applications That Traverse Irregular Data Structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/CGO.2013.6494989

[33] Paul K. Romano, Nicholas E. Horelik, Bryan R. Herman, Adam G. Nelson, Benoit Forget, and Kord Smith. 2015. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy* 82 (2015), 90 – 97. https://doi.org/10.1016/j.anucene.2014.07.048 Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, {SNA} + {MC} 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.

[34] N. Rotem and Y. Ben Asher. 2014. Block Unification IF-conversion for High Performance Architectures. *IEEE Computer Architecture Letters* 13, 1 (Jan 2014), 17–20. https://doi.org/10.1109/L-CA.2012.28

[35] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. 2017. Simplification and Runtime Resolution of Data Dependence Constraints for Loop Transformations. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3079079.3079098

[36] Jaewook Shin. 2007. Introducing Control Flow into Vectorized Code. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07)*. IEEE Computer Society, Washington, DC, USA, 280–291. https://doi.org/10.1109/PACT.2007.41

[37] Jaewook Shin, Mary Hall, and Jacqueline Chame. 2005. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*. IEEE Computer Society, Washington, DC, USA, 165–175. https://doi.org/10.1109/CGO.2005.33

[38] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. 2009. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocessors and Microsystems* 33, 4 (6 2009), 235–243. https://doi.org/10.1016/j.micpro.2009.02.002

[39] Standard Performance Evaluation Corporation (SPEC). 2017. SPEC CPU2017 Benchmark Descriptions.

[40] Shahar Timnat, Ohad Shacham, and Ayal Zaks. 2014. Predicate vectors if you must. In *Workshop on Programming Models for SIMD/Vector Processing*.

[41] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)* (2014).

[42] Christian Wimmer and Hanspeter Mössenböck. 2005. Optimized Interval Splitting in a Linear Scan Register Allocator. In *Proceedings*

of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05). ACM, New York, NY, USA, 132–141. https://doi.org/10.1145/1064979.1064998