Vectorization System for Unstructured Codes with a Data-parallel Compiler IR

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
an der Fakultät für Mathematik und Informatik der
Universität des Saarlandes

eingereicht von

Simon Moll

Saarbrücken, 2021

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	2022-01-25
Dean of the Faculty	Prof. Dr. Thomas Schuster
Chair of the Committee	Prof. Dr. Jan Reineke
First Reviewer	Prof. Dr. Sebastian Hack
Second Reviewer	Prof. Dr. Sven Apel
Academic Assistant	Dr. Andreas Abel

Abstract

With Dennard Scaling coming to an end, Single Instruction Multiple Data (SIMD) offers itself as a way to improve the compute throughput of CPUs. One fundamental technique in SIMD code generators is the vectorization of data-parallel code regions. This has applications in outer-loop vectorization, whole-function vectorization and vectorization of explicitly data-parallel languages.

This thesis makes contributions to the reliable vectorization of data-parallel code regions with unstructured, reducible control flow. Reducibility is the case in practice where all control-flow loops have exactly one entry point.

We present P-LLVM, a novel, full-featured, intermediate representation for vectorizers that provides a semantics for the code region at every stage of the vectorization pipeline. Partial control-flow linearization is a novel partial if-conversion scheme, an essential technique to vectorize divergent control flow. Different to prior techniques, partial linearization has linear running time, does not insert additional branches or blocks and gives proved guarantees on the control flow retained.

Divergence of control induces value divergence at join points in the control-flow graph (CFG). We present a novel control-divergence analysis for directed acyclic graphs with optimal running time and prove that it is correct and precise under common static assumptions. We extend this technique to obtain a quadratic-time, control-divergence analysis for arbitrary reducible CFGs. For this analysis, we show on a range of realistic examples how earlier approaches are either less precise or incorrect.

We present a feature-complete divergence analysis for P-LLVM programs. The analysis is the first to analyze stack-allocated objects in an unstructured control setting.

Finally, we generalize single-dimensional vectorization of outer loops to multi-dimensional tensorization of loop nests. SIMD targets benefit from tensorization through more opportunities for re-use of loaded values and more efficient memory access behavior.

The techniques were implemented in the Region Vectorizer (RV) for vectorization and TensorRV for loop-nest tensorization. Our evaluation validates that the general-purpose RV vectorization system matches the performance of more specialized approaches. RV performs on par with the ISPC compiler, which only supports its structured domain-specific language, on a range of tree traversal codes with complex control flow. RV is able to outperform the loop vectorizers of state-of-the-art compilers, as we show for the SPEC2017 `nab_s` benchmark and the XSBench proxy application.

Zusammenfassung

Mit dem Ausreizen des Dennard Scalings erreichen die gewohnten Zuwächse in der skalaren Rechenleistung zusehends ihr Ende. Moderne Prozessoren setzen verstärkt auf parallele Berechnung, um den Rechendurchsatz zu erhöhen. Hierbei spielen SIMD Instruktionen (Single Instruction Multiple Data), die eine Operation gleichzeitig auf mehrere Eingaben anwenden, eine zentrale Rolle. Eine fundamentale Technik, um SIMD Programmcode zu erzeugen, ist der Einsatz datenparalleler Vektorisierung. Diese unterliegt populären Verfahren, wie der Vektorisierung äußerer Schleifen, der Vektorisierung gesamter Funktionen bis hin zu explizit datenparallelen Programmiersprachen.

Der Beitrag der vorliegenden Arbeit besteht darin, ein zuverlässiges Vektorisierungssystem für datenparallelen Code mit reduziblem Steuerfluss zu entwickeln. Diese Anforderung ist für alle Steuerflussgraphen erfüllt, deren Schleifen nur einen Eingang haben, was in der Praxis der Fall ist.

Wir präsentieren P-LLVM, eine ausdrucksstarke Zwischendarstellung für Vektorisierer, welche dem Programm in jedem Stadium der Transformation von datenparallelem Code zu SIMD Code eine definierte Semantik verleiht.

Partielle Steuerfluss-Linearisierung ist ein neuer Algorithmus zur If-Conversion, welcher Sprünge erhalten kann. Anders als existierende Verfahren hat Partielle Linearisierung eine lineare Laufzeit und fügt keine neuen Sprünge oder Blöcke ein. Wir zeigen Kriterien, unter denen der Algorithmus Steuerfluss erhält, und beweisen diese.

Steuerflussdivergenz induziert Divergenz an Punkten zusammenfließenden Steuerflusses. Wir stellen eine neue Steuerflussdivergenzanalyse für azyklische Graphen mit optimaler Laufzeit vor und beweisen deren Korrektheit und Präzision. Wir verallgemeinern die Technik zu einem Algorithmus mit quadratischer Laufzeit für beliebiege, reduzible Steuerflussgraphen. Eine Studie auf realistischen Beispielgraphen zeigt, dass vergleichbare Techniken entweder weniger präsize sind oder falsche Ergebnisse liefern. Ebenfalls präsentieren wir eine Divergenzanalyse für P-LLVM Programme. Diese Analyse ist die erste Divergenzanalyse, welche Divergenz in stapelallokierten Objekten unter unstrukturiertem Steuerfluss analysiert.

Schließlich generalisieren wir die eindimensionale Vektorisierung von äußeren Schleifen zur multidimensionalen Tensorisierung von Schleifennestern. Tensorisierung eröffnet für SIMD Prozessoren mehr Möglichkeiten, bereits geladene Werte wiederzuverwenden und das Speicherzugriffsverhalten des Programms zu optimieren, als dies mit Vektorisierung der Fall ist.

Die vorgestellten Techniken wurden in den Region Vectorizer (RV) für Vektorisierung und TensorRV für die Tensorisierung von Schleifennestern implementiert. Wir zeigen auf einer Reihe von steuerflusslastigen Programmen für die Traversierung von Baumdatenstrukturen, dass RV das gleiche Niveau erreicht wie der ISPC Compiler, welcher nur seine strukturierte Eingabesprache verarbeiten kann. RV kann schnellere SIMD-Programme erzeugen als die Schleifenvektorisierer in aktuellen Industriecompilern. Dies demonstrieren wir mit dem `nab_s` benchmark aus der SPEC2017 Benchmarksuite und der XSBench Proxy-Anwendung.

Acknowledgements

My thanks go first to my advisor Prof. Sebastian Hack. His advice and the many discussions we had helped shape up the ideas that ultimately condensed in this thesis. I appreciate the independent research environment he provides with the right balance for venturing into new ideas and persisting on hard problems.

I thank Prof. Sven Apel for taking the time to review my thesis. I extend my thanks to the chair of the review committee Prof. Jan Reineke and the academic assistant Dr. Andreas Abel. All of the above made it possible to have a hybrid defense amidst the pandemic.

My time at the lab would have been half as enjoyable without my colleagues and office mates. Thanks, guys, for keeping up the spirit and thanks to Fabian for letting me run experiments on his latest toys.

Finally, thank you, Susanne, for being the best partner I could wish for through the ups and downs of it all.

Contents

Chapter 1.

Introduction

With Dennard Scaling coming to an end, the Single Instruction Multiple Data (SIMD) paradigm offers a way to extract more compute throughput at the limits of contemporary transistor technology. SIMD instructions apply one operation to each scalar element of vector, implementing a restricted form of parallel compute. SIMD ISAs are a steady presence being available from mobile platforms (e.g. ARM NEON [Reddy, 2008], ARM MVE [ARM]), through general-purpose CPUs (e.g. AltiVec [Fuller, 1998],AVX512 [Intel]) to dedicated high-performance hardware (e.g. SVE [Stephens et al., 2017], NEC SX-Aurora TSUBASA [Komatsu et al., 2018], AMD GPUs AMD [2019] or the RISC-V (V extension) [Alon Amid et al., 2019]).

SIMD exists foremost because it is efficiently implementable in hardware not for ease of programmability. Expert SIMD programmers leverage compiler-builtin functions to tap into the compute potential of SIMD. This is a tedious programming task that results in highly target-specific codes. Against the backdrop of stagnating scalar performance, increasingly non-expert programmers rely on SIMD code generators and language support to meet application performance goals. One fundamental paradigm in these tools is data-parallel vectorization, which surfaces in compilers for workitem-centric programming languages (CUDA, OpenCL), outer-loop vectorization or whole-function vectorization.

Data-parallel vectorizers broadly fall into three categories: First, domain-specific SIMD code generators vectorize within the strict limits of their domain, e.g. tree traversal codes [Jo et al., 2013; Ren et al., 2015] or FFTs [Frigo and Johnson, 2005]. Second, compilers for high-level languages for vectorization, such as ISPC [Pharr and Mark, 2012] or Sierra [Leißa et al., 2014], vectorize codes only in their structured language. Finally, the vectorizer passes of general-purpose compilers build on the compilers' unstructured intermediate representations (IR). Compiler IRs are principally for sequential code and have only a rudimentary data-parallel interpretation.

This thesis presents the Region Vectorizer (RV), its underlying program representation and novel analyses and transformations. RV advances the state of the art in the vectorization of data-parallel programs with arbitrary, reducible control flow.

This thesis is structured as follows: Chapter 2 clarifies the terminology and notation used throughout this thesis, along with an introduction to the intermediate representation of the LLVM compiler framework. Chapter 3 gives an introduction to the RV vectorization system. Chapter 4 to Chapter 10 describe the main components of the RV vectorization system. Chapter 11 presents TensorRV, which extends RV to the tensorization of multi-dimensional loop nests. Chapter 12 discusses related work in vectorization frameworks from the angle of full system capabilities, Chapter 13 presents evaluation results. We draw conclusions and give an outlook on future work in Chapter 14.

1.1. Contributions

This thesis makes the following contributions:

- *P-LLVM* - a data-parallel extension of LLVM IR with a formally defined, lock-step semantics (Chapter 4 and Chapter 5). The P-LLVM representation includes key features required in vectorization, such as per-block predication and horizontal operations. The behavior of a well-formed P-LLVM program is predictable and defined even for horizontal operators in divergent control-flow and non-trivial predication. Prior data-parallel IRs are either less expressive, e.g. do not consider predication, not as rigorously defined, e.g. NVIDIA PTX, or do not allow unstructured programs, e.g. ISPC, Sierra. Each ephemeral P-LLVM program in the vectorizer pipeline has a defined semantics, turning the vectorization process into a progression of combinable passes.

- *Partial Linearization* - a new and simple algorithm for partial if-conversion with linear running time and proved properties (Chapter 8). Partial linearization does not insert branches or blocks and gives strong guarantees on retained branches and control dependences. Prior techniques do not give these guarantees for unstructured control flow.

- P-LLVM and partial linearization simplify existing transformations for divergent control flow. We show this for the transformation of divergent loops into uniform loops and the insertion of all-false mask tests (Chapter 9).

- A novel divergence analysis for P-LLVM programs (Chapter 6). This divergence analysis improves over prior work by considering all divergence effects in P-LLVM programs, whether they arise from non-uniform branching or predication. This is the first divergence analysis that can prove the uniformity of stack-objects in predicated, unstructured control flow. We also present an expressive analysis lattice for divergence analysis, which captures stride and alignment at the same time (Chapter 10).

- The control-divergence analysis, a new algorithm to compute control-induced divergence effects in reducible, unstructured control-flow graphs (Chapter 7). We prove correctness, precision and optimal running time for directed acyclic graphs and extend the algorithm to reducible control-flow graphs. This is the first control-divergence analysis that models divergent loops with mixed divergent and uniform exits. We show how prior techniques are less precise or deliver invalid results on a range of realistic examples.

- Tensorization - the generalization of vectorization to multiple dimensions (Chapter 11). Tensorization exposes opportunities for reuse of loaded values. Experiments show speed ups by up to 45% on stencil codes and up to 511% on matrix transpose when loop nest tensorization is combined with a data layout transformation.

- All of the aforementioned techniques are implemented in the Region Vectorizer (RV) vectorization system and evaluated on benchmarks on four different platforms (Chapter 13). The benchmarks comprise a set of tree traversal codes and two benchmark applications, the nab_s benchmark of SPEC2017 and the XSBench proxy application of OpenMC, a neutronics simulation code. In outer-loop vectorization, RV outperforms the best of the Intel C Compiler, GCC and the AMD AOCC compiler on nab_s by 9.2% and on XSBench by 27.8%. On the tree traversal codes, RV vectorizes C++ implementations that match expert implementations in the ISPC language, without requiring any of the annotations that are necessary for ISPC.

1.2. Publications

This thesis is based on the following publications:

1. S. Moll and S. Hack. *Partial Control-Flow Linearization.* In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, [Moll and Hack, 2018].

2. S. Moll, S. Sharma, M. Kurtenacker, and S. Hack. *Multi-dimensional Vectorization in LLVM.* In Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing, WPMVP'19, [Moll et al., 2019].

3. M. Haidl, S. Moll, L. Klein, H. Sun, S. Hack, and S. Gorlatch. *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@SC 2017, [Haidl et al., 2017].

Other Publications

The following results were published in the course of research but do not form part of this thesis:

1. S. Moll, J. Doerfert, and S. Hack. *Input Space Splitting for OpenCL.* In Proceedings of the 25th International Conference on Compiler Construction, CC 2016, [Moll et al., 2016].

2. A. Pérard-Gayot, R. Membarth, P. Slusallek, S. Moll, R. Leißa, and S. Hack. *A Data Layout Transformation for Vectorizing Compilers.* In Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, [Pérard-Gayot et al., 2018].

We enhanced the algorithms presented in Chapter 7 after the research period for this thesis. The improved algorithms are published in the following article and do not form part of this thesis:

1. J. Rosemann, S. Moll, and S. Hack. *An abstract interpretation for SPMD divergence on reducible control flow graphs.* In Proc. ACM Program. Lang. 5, POPL 2021, [Rosemann et al., 2021].

Theses by Students

The following students' theses were advised in the course of the research for this dissertation:

- Thorsten Klößner. "A Transformer-Generic Vectorization Analysis", Bachelor Thesis.

- Dominik Montada. "naTIVE: Target-Independent Vector-Code Generator for Existing Vectorizers", Master Thesis.

- Shrey Sharma. "Multidimensional Auto-Vectorization of Stencil Codes", Master Thesis.

The following thesis fell in the research period but is not related:

- Matthias Kurtenacker. "On Synchronization in the Polyhedral Model", Master Thesis.

Chapter 2.

Background

This chapter introduces the LLVM intermediate representation, the notion of the control-flow graph and loops. We also present auxiliary functions and provide some clarification on the notation that we will use in the subsequent chapters.

2.1. LLVM

LLVM [Lattner and Adve, 2004] is a compiler framework and intermediate representation (IR). Figure 2.1 shows a language reference for the LLVM IR language as we use it in this thesis. The syntax deviates slightly from standard LLVM IR to enhance legibility: We use `elemptr` to mean the address computation instruction (`getelementptr` in standard LLVM IR) and add an explicit `not` instruction. Also we take the liberty of omitting types where they can be inferred or do not matter in the context.

LLVM IR uses Static Single Assignment Form (SSA) [Alpern et al., 1988]. That is every value-yielding instruction assigns its result to a variable that is unique in the function that contains the instruction.

The set $\mathcal{V}$ refers to all values that a LLVM variable can attain. The set should be interpreted in the type context. That is, if a variable `x` has type `fp64` and $y \in \mathcal{V}$ refers to a valuation of `x` then $\mathcal{V}$ is the set of all `fp64` values (**`double`** floating-point type). We use the symbol $\top \in \mathcal{V}$ to refer to the special *undefined value* of LLVM. The undefined value is a wildcard value that is equivalent to every value. It is used to indicate that the result of an operation is not defined. An operation on $\top$ again yields $\top$. Compiler transformations may replace undefined values with any other value, enabling aggressive program transformations.

Sign-agnostic integer arithmetic	
`add/sub` x y	Addition/subtraction
`mul` x y	Multiply
Signed/Unsigned arithmetic	
`sdiv/udiv` x y	Division
`srem/urem` x y	Remainder
Boolean arithmetic	
`or/and/xor` x y	Bitwise or/and/xor
`not` x	Bitwise not
Floating point arithmetic	
`fadd/fsub` x y	Fp addition/subtraction
`fmul/fdiv` x y	Fp multiplication/division
Comparison	
`fcmp` P_{fp} x y	Floating-point compare
$P_{fp} \in \{$ `oeq,one,olt,ole,ogt,oge,ueq,une,ult,ule,ugt,uge` $\}$	
`icmp` P_{int} x y	Integer compare
	$P_{int} \in \{$ `slt,sle,sgt,sge,ult,ule,ugt,uge` $\}$
Memory	
`alloca` *typ*	Stack allocation
`elemptr` P $\mathrm{I}_0 \ldots \mathrm{I}_{n-1}$	Address computation
`load` P	Load value from address P
`store` P V	Store value of V to address P
Other	
`select` C A B	A if C else B
$F(\mathrm{A}_0, \ldots, \mathrm{A}_{n-1})$	Call of function F

(a) Instructions

SSA	
$\phi\ [v_0,\ B_0]\ ..\ [v_n,\ B_n]$	Value v_i from block B_i
Terminators	
`switch` v $[c_i,\ B_i]$, ..	Go to B_i if $v = c_i$
`default(`B_d`)`	Otw, got to B_d.
`br` c A B	Go to A if c else B
`ret` v	Return with value v

(b) Other Primitives

Scalar Types	
void	Type without value
i*n*	*n* bit integer type
fp*n*	*n* bit IEEE 754 binary float
Compound Types	
*typ**	Pointer type
$\{\ typ_0, .., typ_{n-1}\ \}$	Struct type
typ[*n*]	Array type with *n* elements
<*n* x *typ*>	Vector type with *n* elements

(c) Types

Figure 2.1.: LLVM IR reference (dialect).

2.2. Control-Flow Graph

A *Control-Flow Graph* (CFG) $G = (\mathtt{V}, \mathtt{E}, \mathit{entry})$ consists of basic blocks $v \in \mathtt{V}$, control-flow edges $(b, i, s) \in \mathtt{E}$ and a designated $\mathit{entry} \in \mathtt{V}$. If $(b, i, s) \in \mathtt{E}$ then s is the i-th successor of the terminator in b. We require that every block $v \in \mathtt{V}$ is reachable from *entry*. Each basic block $b \in \mathtt{V}$ consists of a list of ϕ nodes, followed by a list of instructions and ends in a terminator.

Control-Flow Edges and Paths. We use the notation $b \rightarrow s \in \mathtt{E}$ to mean $\exists i.(b, i, s) \in \mathtt{E}$. Likewise, we denote by $\pi \in a \rightarrow^{+} b$ a non-empty path π from a to b through a chain of edges (transitive closure of the control-flow edges). The notation $\pi \in a \rightarrow^{*} b$ means that either π is the empty path and $a = b$ or $\pi \in a \rightarrow^{+} b$ (reflexive and transitive closure of the control-flow 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 $a \in \mathtt{V}$. We will say two paths $\pi_1 \in a \rightarrow^{*} x$ and $\pi_2 \in a \rightarrow^{*} y$ are *almost node-disjoint*, if they do not share any node except a. Two paths $\pi_1 \in a \rightarrow^{*} z$ and $\pi_2 \in a \rightarrow^{*} z$ are called *inner-node disjoint*, if $\pi_1 \neq \pi_2$ and they have no common nodes except a and z.

A CFG that contains no path $a \rightarrow^{+} a$, for any $a \in \mathtt{V}$, is a *Directed Acyclic Graph* (DAG).

2.3. CFG Properties

Dominance and Post-Dominance. In a graph G, the block $a \in \mathtt{V}$ is said to *dominate* [Prosser, 1959] $b \in \mathtt{V}$ (a is a dominator of b), written $a \succeq^{D} b$, iff every path $\pi \in \mathit{entry} \rightarrow^{*} b$ contains a. Symmetrically [Cytron et al., 1991], the block $a \in \mathtt{V}$ is said to *post dominate* $b \in \mathtt{V}$ (a is a post dominator of b), written $a \succeq^{PD} b$, iff every complete path $\pi \in b\downarrow$ contains a.

Control Dependence. A block $k \in \mathtt{V}$ is *control dependent* on an edge $a \rightarrow b \in \mathtt{E}$, iff $k \succeq^{PD} b$ and $k \not\succeq^{PD} a$. We use the notation $\mathit{cdep}(k) \subseteq \mathtt{E}$ to denote the set of all edges $a \rightarrow b \in \mathtt{E}$ that $k \in \mathtt{V}$ is control dependent on [Ferrante et al., 1987; Cytron et al., 1991]. The set of blocks that $k \in \mathtt{V}$ is control dependent on is defined as $\mathit{cdepB}(k) = \{\, a \in \mathtt{V} \mid \exists b.a \rightarrow b \in \mathit{cdep}(k) \,\}$ The inverse, the set of blocks that are control dependent on b, is given by $\mathit{cdep\_on}(b) = \{\, k \in \mathtt{V} \mid b \in \mathit{cdepB}(k) \,\}$

2.4. Reducibility and Loops

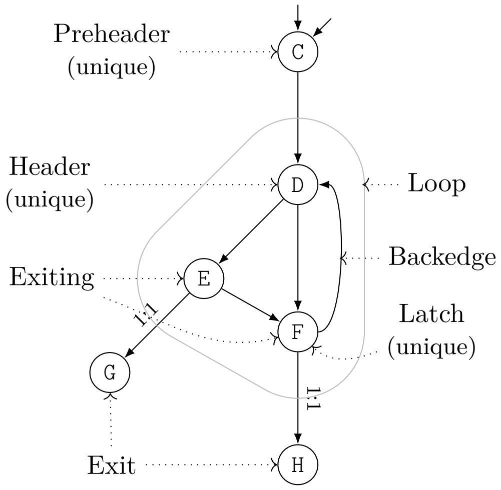


Figure 2.2.: Anatomy of a natural loop.

Throughout this thesis, we shall assume that CFGs are reducible [Hecht and Ullman, 1972; Aho et al., 1986]. A reducible CFG is defined as follows [Hecht and Ullman, 1974]: A depth-first search (DFS) on CFGs yields a set of retreating edges, i.e. edges $x \to y \in \mathtt{E}$ such that y is an ancestor of x in the depth-first search tree. In a reducible CFG, all DFSs yield the same set of retreating edges and for every retreating edge $x \to y \in \mathtt{E}$ it holds that y dominates x. A retreating edge in a reducible CFG is called a *back edge*. We denote by $\mathit{Backedges} \subseteq \mathtt{V}$ the set of all back edges.

Every back edge defines a so-called *natural loop*. Figure 2.2 shows an example of a natural loop and the naming convention for its parts. For every $x \to y \in \mathit{Backedges}$, we call $x \in \mathtt{V}$ a *latch block* and $y \in \mathtt{V}$ a *loop header*. The natural loop of a back edge $x \to y \in \mathit{Backedges}$ is the set if blocks that contains both x and y and all $z \in \mathtt{V}$ such that z reaches the latch block x without going through the loop header y. Natural loops induce a loop tree through subset inclusion, a loop whose blocks are subset of the blocks of another loop is a child loop of the other loop. We define *Loops* as the set of a all loops in the CFG.

We use the following formal notation. In the following, let $L \in \mathit{Loops}$ be a natural loop. We call $b \in \mathtt{V}$ a *loop exit* of the loop L if $b \notin L$ and there is a control-flow edge $a \to b \in \mathtt{E}$ such that $a \in L$. Analogously, we call $a \in \mathtt{V}$ an *exiting block* of the loop L if $a \in L$ and there is a control-flow edge $a \to b \in \mathtt{E}$ such that $b \notin L$. *lp B* is the inner-most loop that contains the basic block B. *lplatch L* denotes the unique latch block of the loop L, *lpexits L* denotes the set of exit blocks of the loop L. *lphead L* is the unique loop header of the loop L, We require that every loop header has exactly one predecessor that is not part of the loop. This block is called the *preheader* of the loop.

We make the following structural assumptions on loops for sake of simplicity: We require that all back edges are disjoint, i.e. there are no two back edges $x \to y \in \mathit{Backedges}$ and $x' \to y' \in \mathit{Backedges}$ such that $y = y'$. We require that $\forall a \in \mathtt{V}.a{\downarrow} \neq \emptyset$, that is all loops have exits. We assume that there is a one-to-one correspondence between loop exiting and exit blocks. These properties can be established in any reducible CFGs by the following means: If there are multiple back edges for the same loop header, insert a new block that branches to this loop header and make all former latch blocks branch to that block instead of the loop header. This new block is then a unique latch block. Uniqueness of the pre-header can be achieved similarly by funneling all edges that enter the loop header from outside the loop through a new dedicated basic block. The one-to-one constraint for loop exit and exiting blocks only requires splitting the loop exiting edges.

We require Loop-Closed SSA form [Pop, 2006] to simplify the algorithms, e.g. the divergence analysis. A program is in LCSSA form if all loop live outs are funneled through ϕ nodes in the loop exits. Variables that are defined in loops and used outside of their defining loop are thus easily identified by inspecting the LCSSA ϕ nodes in the loop exits. Compiler support for LCSSA form is common and available, for example, in GCC and LLVM. LLVM includes passes that establish LCSSA form in LLVM IR programs.

2.5. Block Index

The function *BlockIndex* $\in \mathbb{V} \rightarrow \mathbb{N}$ is the index of a block in a toposort (ignoring back edges). We call such an index *compact* with regards to some set $B \subseteq \mathbb{V}$, if $\mathit{BlockIndex}(B)$ is equal to an interval $[a, b]$ for some $a, b \in \mathbb{N}$. That is there are no block indices in the interval whose blocks are not in B and the block index of all elements of B is within that interval. A block index is a topological block index that is compact with regards to all blocks of the CFG, all loops and all dominated block set of the CFG [Wimmer and Mössenböck, 2005]. Every loop header has the minimum index of the blocks in the loop. Since we impose loop latches to be unique, every loop latch has the maximum index of the loop blocks.

2.6. Mathematical Notation and Conventions

The function gcd $\in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N}$ computes the *greatest common divisor* of two numbers. For the case $a \neq 0$ and $b \neq 0$, we define $\gcd(a, b)$ as the maximal $k \in \mathbb{N}$ such that there exist $a', b' \in \mathbb{Z}$ with $ka' = a$ and $kb' = b$. We extend the definition of gcd to 0 by letting $\gcd(0, a) = \gcd(a, 0) = |a|$.

We denote by $q|n$ that $q \in \mathbb{Z}$ divides $n \in \mathbb{Z}$.

We use the notation $[x_1, .., x_n]$ to construct a vector value where x_i is the i-th element value. We interchangeably use functions over element indices to define vector values, that is

$$\lambda x \in \{ 0, 1, 2, 3 \} .x^2 \qquad \text{is equivalent to} \qquad [0, 1, 4, 9]$$

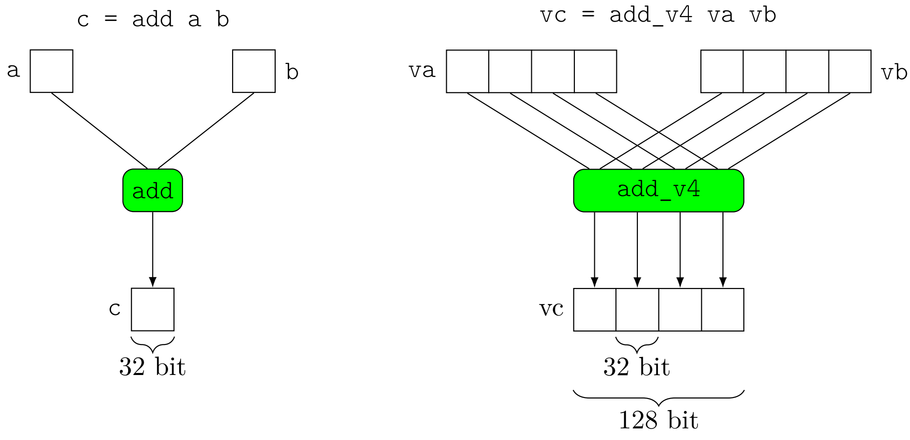
The map^n function applies a function f element-wise to all vector arguments that follow f in the argument list. Non-vector arguments are implicitly broadcast to a vector of size n. An example:

$$\mathit{map}^n\ f\ x^n\ y\ z^n = [f(x_0, y, z_0), .., f(x_{n-1}, y, z_{n-1})]$$

2.7. Single Instruction Multiple Data

Single **I**nstruction **M**ultiple **D**ata (SIMD) [Flynn, 1972] instructions apply one operation on multiple data in parallel. In a typical SIMD ISA, we expect to find a matching SIMD instruction for every scalar instruction that operates on scalar integer or floating point data. Figure 2.3 shows a scalar `add` and its SIMD version, `add_v4`.

Scalar instructions, for example the `add` instruction in Figure 2.3a, operate on scalar input registers and produce a scalar result. The example shows a 32 bit add operation on 32 bit scalar CPU registers. The matching SIMD instruction, `add_v4`, is shown in Figure 2.3. Instead of operating on scalar registers, `add_v4` operates on SIMD registers, which are 128 bit wide in this case. In case of the `add_v4` the 128 bit of the operand and destination register are interpreted as 4×32 bit. We refer to these logical elements of a SIMD register as *SIMD lanes*. How many SIMD lanes are available for an instruction depends on the element size of the SIMD operation. For example, if a SIMD register has 128 bit, then a SIMD add of 64bit elements only produces two results in one invocation. This logic subdivision transfers to SIMD instructions. When we say that an instruction operates on a certain lane i, we mean that the SIMD instruction reads the i-th element of every SIMD operand and writes the result to the i-th element position of the destination register.



(a) Scalar instruction.

(b) SIMD version of the same instruction.

Figure 2.3.: SIMD instruction archetype.

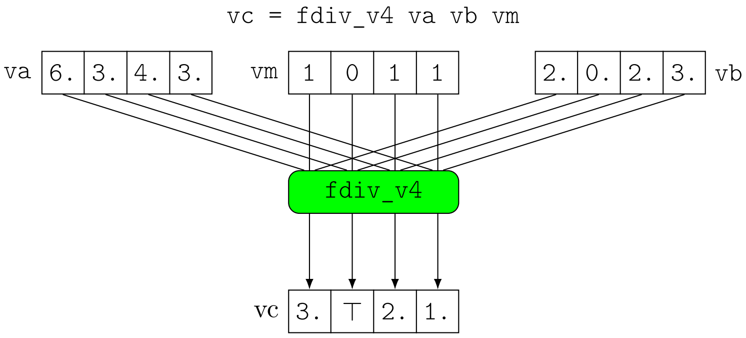


Figure 2.4.: Predicated floating-point division for SIMD width 4.

For sake of clarity, we use to the following naming scheme for SIMD instructions. We suffix a scalar operation with `_v`W to indicate that we are using the SIMD version of it with W many lanes. For example, a scalar addition is given by `add`. The SIMD variant for width 4 is given by `add_v4`.

Other SIMD Instructions. Some typical SIMD instructions deviate from the archetypical pattern in Figure 2.3. The `broadcast_v`W`(v)` instruction, takes a scalar value `v` and replicates it into the lanes of a SIMD register. For sake of legibility, we omit explicit broadcasts in code listings. When a scalar operand is used in a SIMD operation, the broadcast of that operand is implied. Almost all SIMD architectures support vector memory loads and stores. A SIMD load `load_v`W`(ptr)`, loads a full SIMD register from the address given by `ptr`. The SIMD store `store_v`W`(ptr, v)` symmetrically stores the full SIMD register `v` to the address in `ptr`.

Among the regular SIMD instructions, we highlight the *select* operation. The operation `select_v`W`(c, a, b)` is an element-wise select of values from two vectors `a` and `b`, depending on a bit vector `c`. The result lanes are equal to `a` where the bit in `c` is true. Otherwise, the result lane holds the value of the lane in `b`.

Predicated SIMD Instructions. Many recent SIMD ISAs support predicated SIMD instructions. Predicated SIMD instructions have an additional predicate operand, which is a bit SIMD vector. The operation is only performed on those lanes where the mask bit is `1`. Consider the SIMD builtin function for predicated floating-point division in Figure 2.4. The mask bit on the second lane is `0` and so there is no *division by zero*.

SIMD ISAs give varying guarantees for values on masked-off lanes For example, the AVX512 `vdivpd` instruction (floating-point division) comes in two forms that either preserve the value of the destination register or yield `0` on masked-off lanes. Other ISAs do not support SIMD predication at all (i.e. ARM NEON) or only for some instructions (there is no SIMD integer division in AVX512).

The compiler needs to define a semantics for the masked-off lanes in the predicated SIMD instructions of the IR. In order to not tailor these towards a specific SIMD ISA, we define that the result on masked-off lanes is the undefined value ($\top$). Compiler backends for non-predicating SIMD ISAs can simply ignore the predicate for side-effect-free operations. Lanes holding $\top$ can be swapped for defined values using a SIMD `select_v`W. For example, the AVX512 backend may emit a *zeromasking* `vdivpd` instruction, matching a `fdiv_v4` plus `select_v4` for `0` on masked-off lanes.

Chapter 3.

Motivation and System Overview

This chapter gives an overview of the P-LLVM program representation and the RV vectorization system that is built on it. During each section, we shed light on a challenge in vectorization and extend the vectorizer to account for it.

Section 3.1 starts with a basic vectorizer for outer loops that are known to be parallel. Section 3.2 introduces the concept of uniformity. In Section 3.3, we extend the vectorizer pipeline to support Whole-Function Vectorization through the concept of regions and SIMD lane threads. Section 3.4 demonstrates how partial linearization is integrated into the system, giving rise to predication in the IR. Section 3.5 introduces the concept of horizontal operations, lock-step execution and runtime schedules. Section 3.6 discusses the transformation of divergent loops, exposing the semantic difference between SIMD code and lock-step execution.

We conclude in Section 3.7 with the final pipeline of the *Region Vectorizer*, handing over to individual chapters for the technical presentation.

3.1. Parallel-Loop Vectorization

Some loops can be vectorized simply by rewriting them with SIMD intrinsics. The loop in Figure 3.1a is an example of this. It is *parallel*, meaning that no iteration writes to an address that another iteration writes to or reads from. And, importantly, its trip count happens to be the SIMD width (assuming we are vectorizing for a 256 *bit* SIMD ISA). Such a loop is vectorized by replacing every scalar instruction with a SIMD instruction. We call this very mechanical process *Widening*. After all instructions have been vectorized, we remove the loop around them, resulting in the code shown in Figure 3.1b. One execution of the SIMD instructions does the work of the former scalar loop.

```
double * A, * B, * C;
// ...
for (int i = 0; i < 4; ++i) {
  b = B[i];
  c = C[i];
  a = b + c;
  A[i] = a;
}
```

(a) Parallel loop (`A`, `B` and `C` assumed not to alias). The loop trip count is the SIMD width.

```
double * A, * B, * C;
// ...
int i = 0;
b = load_v4(&B[i]);
c = load_v4(&C[i])
a = add_v4(b + c);
store_v4(&A[i], a);
```

(b) *Widened* loop with SIMD instructions.

Figure 3.1.: Widening a constant trip count loop.

Loop Vectorizers automate this process as a transformation in the compiler. In the context of this thesis, we use the LLVM [Lattner and Adve, 2004] compiler framework. The intermediate representation of LLVM (LLVM IR) is based on Control-Flow Graphs. Nevertheless, we stick to code listings for the purpose of legibility where it is appropriate. The transformations and observations on those listings can be made equivalently on the actual IR.

```
for (int i = 0; i < n; ++i) { // parallel
  double a = 0.0;
  for (int j = 0; j < m; ++j) {
    double v = A[j*m + i];
    a += v;
  }
  B[i] = a;
}
```

(a) Scalar loop.

```
for (int i = 0; i < n; i += 4) {
  // for (int l = 0; l < 4; ++l) {
    double4 a = [0.0, 0.0, 0.0, 0.0];
    for (int j = 0; j < m; ++j) {
      double4 v = load_v4(&A[j*m + i]);
      a = add_v4(a, v);
    }
    store_v4(&B[i], a);
  // }
}
// [..] remainder loop
```

(b) SIMD loop after widening.

```
int minedI;
// strip-mined loop
for (minedI = 0; (minedI + 3) < n; minedI += 4)
{
  for (int l = 0; l < 4; ++l) {
    int i = minedI + l;
    double a = 0.0;
    for (int j = 0; j < m; ++j) {
      double v = A[j*m + i];
      a += v;
    }
    B[i] = a;
  }
}

// remainder loop
for (int i = minedI; i < n; ++i) {
  double a = 0.0;
  for (int j = 0; j < m; ++j) {
    double v = A[j*m + i];
    a += v;
  }
  B[i] = a;
}
```

(c) Scalar loop after strip mining.

Figure 3.2.: Strip mining example.

There are two observations that make widening applicable to a wider class of loops. First, the loop vectorizer can employ strip mining to extract SIMD width trip count loops from loops with variable trip counts. Second, widening is possible in the presence of nested loops if one criterion is met: Variables that are defined on the level of the loop to vectorize, that is not in loops inside of it, must not transitively flow into the exit conditions of the inner loops.

Figure 3.2a shows an example for an inner loop whose exit condition is independent of the loop to vectorize in this way. We are looking to vectorize the `i` loop. The `j` loop exits when `j >= m`. Neither `m` nor the start, end or step value of `j` depend on any variable defined in the `i` loop, namely `i` or `a`. Therefore, we can vectorize the `i`-loop and leave the loop iteration variable `j` and its loop exit condition as they are.

Unlike in our first widening example, the trip count of the `i`-loop of Figure 3.2a is not equal to the SIMD width. We employ *strip mining* to split the loop into two nested loops such that the inner one has exactly the SIMD width as its iteration count. We call the two resulting loops the outer, strip-mined loop, and the inner, strip loop. Strip mining results in a third loop that handles the remainder iterations whenever the trip count of the original loop is not a perfect multiple of the SIMD width. For our example, the strip-mined code is shown in Figure 3.2c. The step size of the strip-mined loop in line Line 3 is the SIMD width. The strip loop in Line 5 iterates over all the in-between elements. The strip loop will be vectorized. The remainder loop in Line 17 takes care of all remaining scalar iterations and remains un-vectorized.

Finally, the widening procedure is applied to the strip loop. Every scalar instructions is replaced with the SIMD instruction. The strip loop is afterwards removed from the code, yielding the result shown in Figure 3.2b. At this point, the parallel-loop vectorizer has successfully vectorized the outer loop.

Loop Vectorizer Pipeline. We summarize the phases of a basic parallel-loop vectorizer pipeline in Figure 3.3.

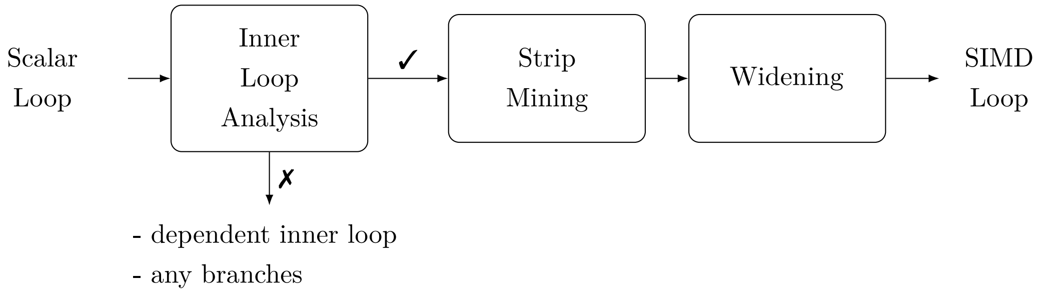


Figure 3.3.: Flow of a basic (parallel) loop vectorizer.

The scalar loop is parallel. The inner loop analysis checks that all loop exit conditions of the inner loops are independent of the parallel loop. If this is not the case or if there are any other branches inside the parallel loop, the system bails and the parallel loop remains un-vectorized. If the exit conditions of the nested loop are independent of the parallel loop and there are no other branches, the system proceeds. The loop is strip mined and the instructions in the strip loop are widened into SIMD code.

3.2. Uniformity and Divergence Analysis

The vectorizer pipeline outlined in Section 3.1 can only handle a very restricted class of control flow: nested loops whose exit conditions are independent of the loop to vectorize. It denies to vectorize loops that do not fall into this category. In the course of this section, we extend the vectorizer to handle a certain class of branches.

```
1  for (int i = 0; i < 4; ++i) { // parallel
2    double a = 0.0;
3    for (int k = 0; k < n; ++k) {
4      bool p = B[k] > 0;
5      if (p) {
6        d = C[k,i];
7        a += d;
8      }
9    }
10   A[i] = a;
11 }
```

(a) Scalar loop with uniform branch.

```
1  // for (int i = 0; i < 4; ++i) {
2    double4 a = 0.0;
3    for (int k = 0; k < n; ++k) {
4      bool p = B[k] > 0;
5      if (p) {
6        double d = load_v4(&C[k,i]);
7        a = add_v4(a, d);
8      }
9    }
10   store_v4(A[i], a);
11 // }
```

(b) Vectorized loop.

Figure 3.4.: Widening works if branch conditions are *uniform*.

Consider the vectorization of the loop in Figure 3.4a. Due to the branch in Line 5 our basic vectorizer will refuse to vectorize the loop. There is no SIMD branch instruction. However, we can enhance the widening process to generate scalar branch instructions in some cases. This improved widening stage leaves branches and instructions scalar if all strip loop iterations and thus SIMD lanes compute the same result. When we proceed with widening despite the branch, widening yields the valid SIMD code shown in Figure 3.4b.

The value of `p` only depends on `k` and is independent of the iteration variable `i`. Thus, the widening stage can keep `p = B[k] > 0` and the branch on `p` scalar.

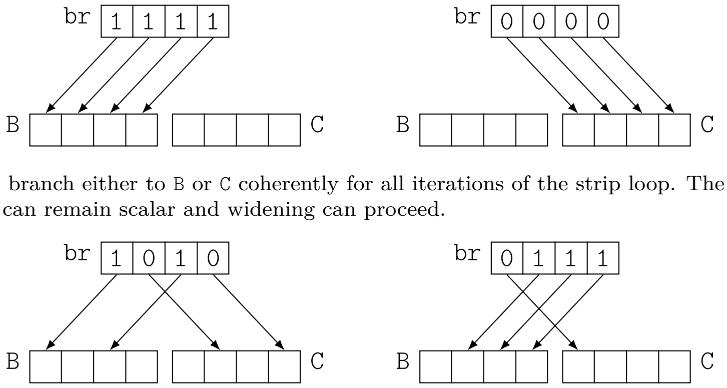
Uniformity. It is possible to vectorize code with branches if the branches are *uniform*. In a uniform branch, the branching decision is the same for all iterations of the strip loop as shown in Figure 3.5a. Therefore, the vectorizer can leave uniform branches scalar. A non-uniform branch is called *divergent* and a non-uniform instruction *varying*. In a divergent branch, the branch condition may evaluate differently for different iterations of the strip loop. This is shown in Figure 3.5b. It is not possible to widen a divergent branch because SIMD ISAs do not support branching to multiple blocks at once. Therefore, the system needs to verify that all branches are uniform before attempting to actually vectorize the loop.

Divergence Analysis. We add a *Divergence Analysis* stage to the vectorizer to detect divergent branches. The divergence analysis is an analysis that performs widening in the abstract without generating code. It classifies each instruction and branch as *uniform* (`u`) or otherwise as *varying/divergent* (`v`).

The iteration variable of the loop to vectorize loop is *varying*. All other values are initially assumed to be *uniform*. The divergence analysis passes over the scalar code classifying every instruction as either *uniform* or *varying*. The divergence analysis will use the same decision logic as the improved widening stage. If all operands of an instructions are *uniform*, then the instruction itself is *uniform*. If a loop contains a divergent branch, it cannot be vectorized. Therefore, Divergence analysis can stop as soon as it encounters the first divergent branch.

Example. Figure 3.6a shows the output of the divergence analysis on an example. In this example, the `i`-loop is vectorized and `i` is the initially known *varying* value. In Figure 3.6b, we show the CFG

```
// for (l = 0; l < 4; ++l) {
   [..]
A: bool p = [..]
   if (p) goto B;
     else goto C;
}
```



(a) If p is *uniform*, the if-statement will branch either to B or C coherently for all iterations of the strip loop. The branch can remain scalar and widening can proceed.

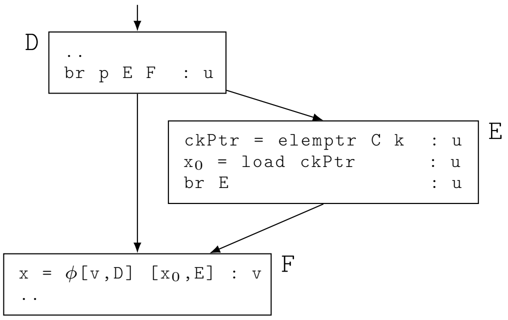
```
// for (l = 0; l < 4; ++l) {
   [..]
A: bool4 p = [..]
   if (p) goto B;
     else goto C;
}
```

(b) If p is *varying*, the branch could go to B and C for different strip loop iterations at the same time. The two outcomes for p to the right are just examples, any combination of B and C is possible, including the outcomes of the uniform case in Figure 3.5a. There is no SIMD branch instruction. Vectorization fails.

Figure 3.5.: Approaching a uniform (3.5a) and a divergent (3.5b) branch in the widening phase.

```
B: for (i = 0; i < n; ++i) {
     v = A[i];                   : v
C:   for (k = 0; k < m; ++k) {
D:     p = B[k];                 : u
       if (p) {                  : u
         x = v;                  : v
       } else {
E:       x = C[k];               : u
       }                 // x  : v
F:     D[k] = x;
     }
   }
```

(a) Program with abstract divergence analysis tags. *Uniform* (u) and *varying* (v).



```
..
br p E F   : u
```

```
ckPtr = elemptr C k   : u
x0 = load ckPtr       : u
br E                  : u
```

```
x = φ[v,D] [x0,E] : v
..
```

(b) Excerpt CFG from Figure 3.6a showing how a ϕ node is classified as *varying*.

Figure 3.6.: Basic divergence analysis run.

of the program in Figure 3.6a tagged with the analysis result of the divergence analysis. There is a ϕ node in block F. In a control uniform CFG, a ϕ node that has only *uniform* incoming values is also *uniform*. The divergence analysis computes the divergence of x by joining the divergence of the incoming values. Since v is *varying*, x is *varying* as well. The only branch condition p is uniform and so the CFG is control uniform. In the widened code, p will remain scalar and the ϕ node x receives a vector value from E and the broadcasted value of v from D.

Vectorizer Pipeline. We add a new *Divergence Analysis* stage to check all branch conditions for *uniformity.* The divergence analysis aborts as soon as *control divergence*, that is any divergent branch, is detected. If the code is control uniform, that is all branches are uniform, we know that widening can proceed keeping all branches scalar.

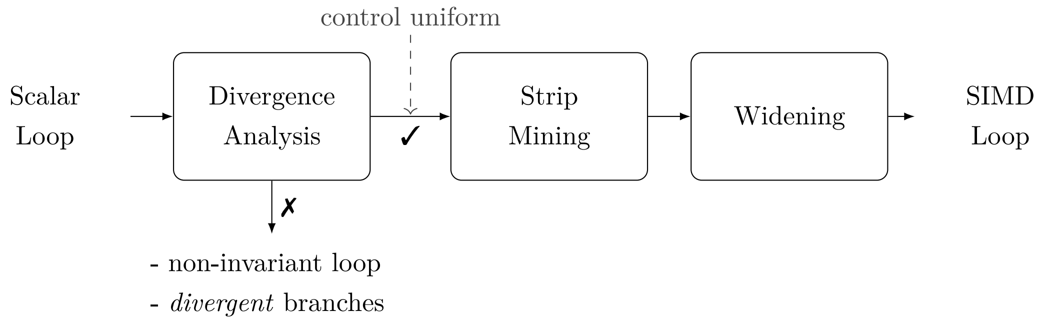


Figure 3.7.: Parallel loop vectorizer for uniform control.

3.3. Region Vectorization

```
float
myPow(float x, int y) {
  int absY = \
    select(y < 0, -y, y);

  float accu = 1.f;
  for (;absY > 1: --absY) {
    accu = accu * x;
  }

  float result = accu;
  if (y < 0) {
    result = 1.0f / accu;
  }
  return result;
}
```

(a) Scalar source function.

```
float4
myPow_vu_v4(float4 x, int y) {
  int absY = \
    select(y < 0, -y, y);

  float4 accu = 1.f;
  for (;absY > 1: --absY) {
    accu = fmul_v4(accu, x);
  }

  float4 result = accu;
  if (y < 0) {
    result = fdiv_v4(1.0f, accu);
  }
  return result;
}
```

(b) Generated SIMD version.

```
float4 r;
r[0] = myPow(x0, y)
r[1] = myPow(x1, y)
r[2] = myPow(x2, y)
r[3] = myPow(x3, y)
```

(c) The scalar `myPow` has to be called for each parameter set.

```
float4 r;
r = myPow_vu_v4([x0, x1, x2, x3], y)
```

(d) The SIMD version computes the results of four parameter sets at once.

Figure 3.8.: Whole-function vectorizers generate `myPow_vu_v4` from `myPow` given only the SIMD declaration.

In the following, we generalize the vectorizer pipeline to vectorize code regions. Code regions make the vectorizer applicable to loop vectorization as well as whole-function vectorization.

Whole-Function Vectorization [Karrenberg and Hack, 2011] is the task of automatically generating a SIMD function from a scalar function. We show an example in Figure 3.8. One invocation of `myPow_vu_v4` returns the same result as four invocations of `myPow`. Whole-function vectorization starts from a scalar function and a declaration for the desired SIMD function. Each parameter of the scalar function either has the same type in the SIMD function or is widened to a vector. In the example of `myPow`, the SIMD declaration reads `myPow_vu_v4(float4, float)`. The first argument `x` is `float` in the scalar function and becomes a `float4` in the SIMD version. The second argument `y` is `float` in the scalar function and stays that way in the SIMD function. This means that the `t`-th lane of the return value of `myPow_vu_x(x, y)` equals `myPow(x[t],y)`. The notation `x[t]` subscripts the `t`-th element of the vector `x`.

We extend the vectorization pipeline to whole-function vectorization through the concept of *Region Vectorization*. In this context, a *Region* is a single-entry, single-exit subgraph (SESE region) of the Control-Flow Graph. Region Vectorization is a generalization of loop and whole-function vectorization.

Basic Region Vectorizer Pipeline

Figure 3.9 shows a basic region vectorization pipeline. Since a region CFG is not necessarily a loop, we can no longer assume that there are iteration variables. We introduce a surrogate concept: SIMD lane threads. Each lane thread is represented by a unique identifier $\mathtt{t} \in \mathcal{T}$, such that each thread uniquely maps to a SIMD lane. Thus, the threads are ordered and form a thread array. With the abstraction of regions and the thread array, whole-function vectorization and loop vectorization both are recast as region vectorization problems.

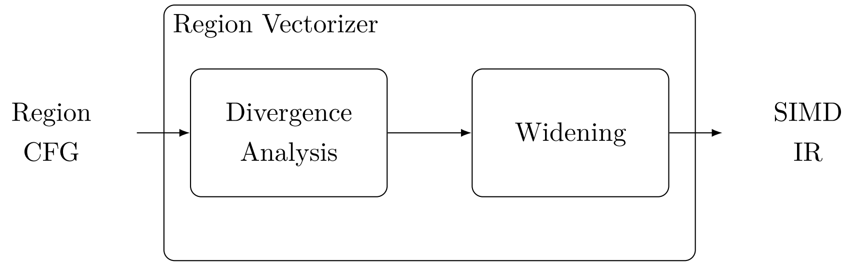


Figure 3.9.: The shared *Region Vectorization* pipeline.

Outer-Loop Vectorization

The loop vectorizer compiler pass is shown in Figure 3.10. In the loop vectorization setting, each iteration of the (conceptual) strip loop maps to one thread. Internally, the loop vectorizer pass defines the *Region CFG* as the body of the strip loop. The pass takes care of building the scalar remainder loop and embedding the SIMD loop body back into the program.

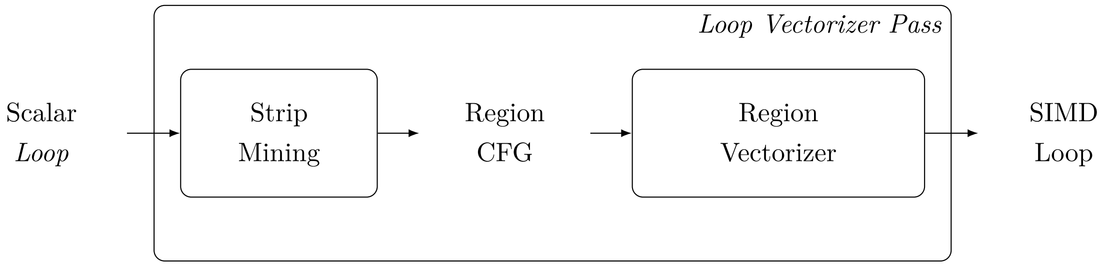


Figure 3.10.: Driver pipeline for *Loop Vectorization*.

Whole-Function Vectorization

The *Whole-function vectorizer Pass* is shown in Figure 3.11. For whole-function vectorization, each thread is associated with one parameter set for the scalar function. The parameter set of thread `t` comprises of `x[t]` for any vector parameter `x` and `y` for any scalar parameter `y` of the SIMD declaration. If a scalar parameter has a SIMD type in the SIMD declaration, then this parameter variable is varying. Otherwise, the parameter variable is uniform. The *Region CFG* is exactly the CFG of the scalar function. The widening phase emits all blocks and SIMD instructions into the SIMD declaration, remapping parameters of the scalar function to those of the SIMD function.

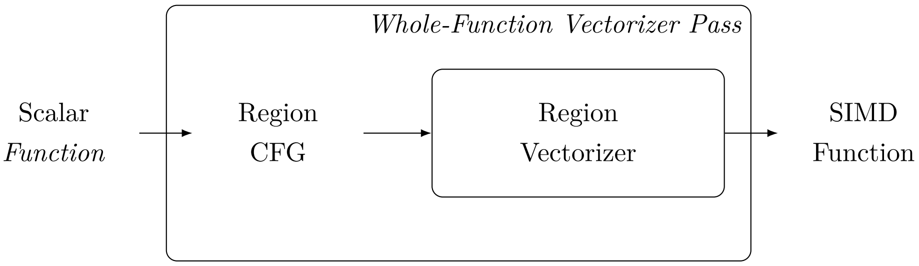


Figure 3.11.: Driver pipeline for *Whole-Function Vectorization*.

3.4. If Conversion and Control-Induced Divergence

The widening stage requires the Control-Flow Graph to be control uniform. This means that the vectorizer pipeline developed up to this point cannot vectorize the parallel loop in Figure 3.12a. This is because the branch condition p is *varying* and thus the branch becomes *divergent*. We present partial linearization, an improved algorithm for if-conversion [Allen et al., 1983], to solve this problem.

```
H: for (int i = 0; i < 4; ++i) {
B:   double v = 2.0;
     bool p = i % 2;
     if (p) {
C:     v = B[i];
     }

D:   C[i] = v;
   }
X:
```

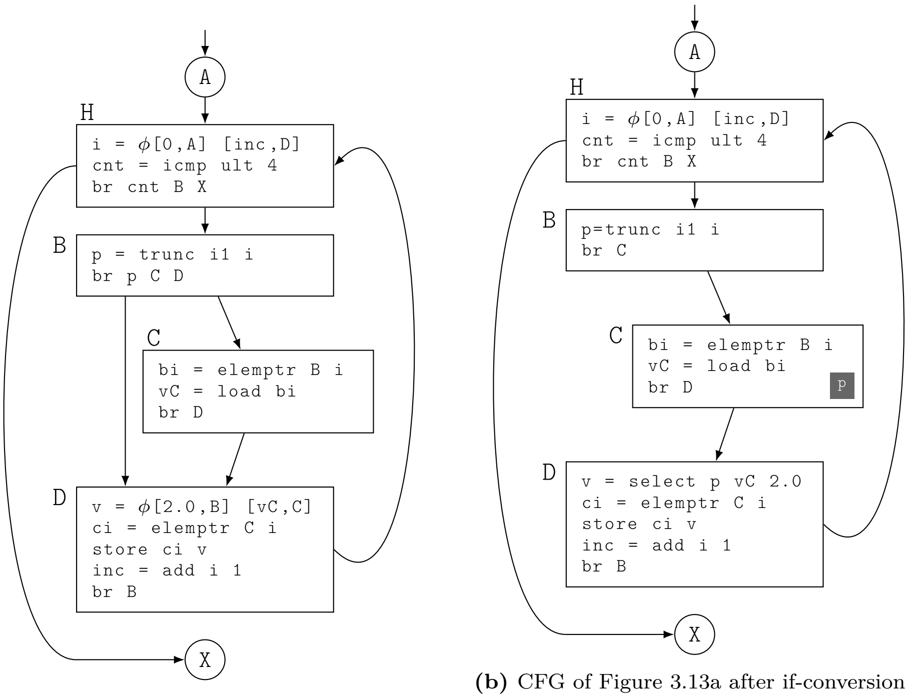
(a) Loop with divergent branch.

```
// for (int i = 0; i < 4; ++i) {

  bool2 p = mod_v4([0,1,2,3], 2);
  // if (p) {
  vC = masked_load_v4(&B[0], p)
  // }
  v=select_v4(p,vC,2.0)
  store_v4(&C[0], v);
// }
```

(b) SIMD code after if-conversion and widening.

Figure 3.12.: Vectorizing with if-conversion.



(a) CFG of Figure 3.12a.

(b) CFG of Figure 3.13a after if-conversion. The gray box indicates that all instructions in block C are predicated by the variable p.

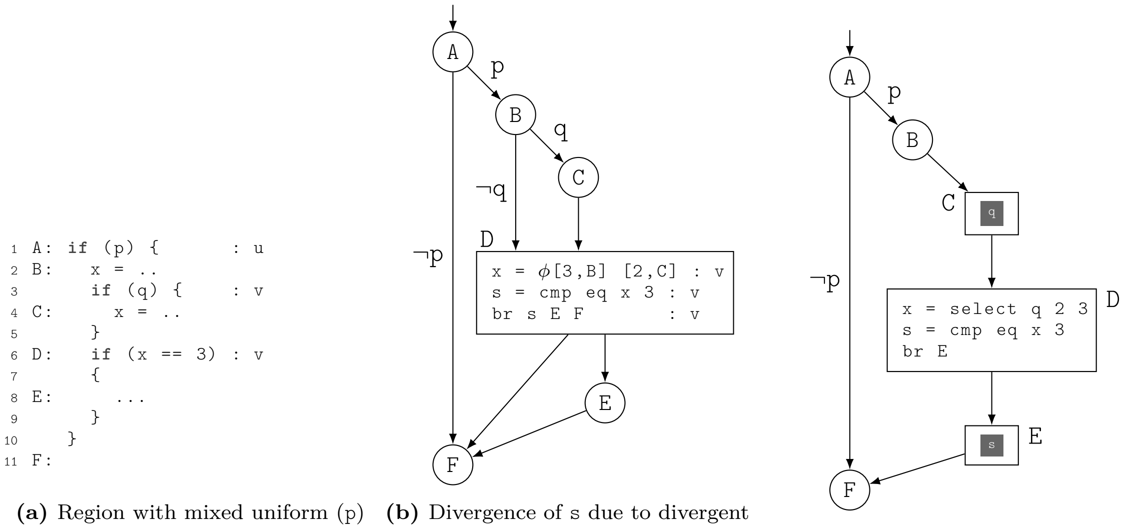
Figure 3.13.: Before and after if-conversion.

If Conversion. If-conversion removes branches from the CFG by unconditionally executing all its successors in sequence. If-conversion adds *predicates* to the code to make sure that the instructions in the if-converted blocks execute under the same condition as in the original code. If a block has a predicate it means that the instructions in the block only execute for those lanes where the block predicate is true.

Consider Figure 3.13a, which shows the CFG of Figure 3.12a. In the original CFG, the block `B` branches divergently to `C` and `D`. Figure 3.13b shows the if-converted version. To annotate block predicates in figures, we show the predicate variable in a gray box inside the block they are predicating. After if-conversion block `B` unconditionally branches to `C`. In the if-converted code, we add `p` as a predicate variable to the block `C`.

In the process of if-conversion ϕ nodes are replaced by explicit *select* instructions that pick a value based on a boolean predicate. For example, the `v =` ϕ node in Figure 3.13a is folded into a `select` node in Line 7 of Figure 3.13b.

Partial Linearization. If-conversion makes the code less efficient as branches are removed that skip over instructions that do not contribute to the computation. The instructions are predicated but still the CPU has to process them. Therefore, it is desirable to if-convert only the divergent branches and leave the uniform branches unmodified. Chapter 9 presents *partial control-flow linearization*, a novel partial if-conversion algorithm. Partial linearization is the first algorithm to give guarantees on retained uniform control flow in unstructured, reducible CFGs. We show an example run in Figure 3.14. Figure 3.14a sketches a code fragment with mixed uniform and divergent branches. We assume that `p` is uniform and that `q` is varying. Figure 3.14b shows the CFG of the code and Figure 3.14c the code after partial linearization. The uniform branch in `p` is retained and only the divergent branch in `q` is if-converted.



```
A: if (p) {         : u
B:   x = ..
     if (q) {       : v
C:     x = ..
     }
D:   if (x == 3)  : v
     {
E:     ...
     }
   }
F:
```

```
x = ϕ[3,B] [2,C] : v
s = cmp eq x 3 : v
br s E F         : v
```

```
x = select q 2 3
s = cmp eq x 3
br E
```

(a) Region with mixed uniform (`p`) and divergent (`q`) branches.

(b) Divergence of `s` due to divergent control from `B`.

(c) After partial linearization.

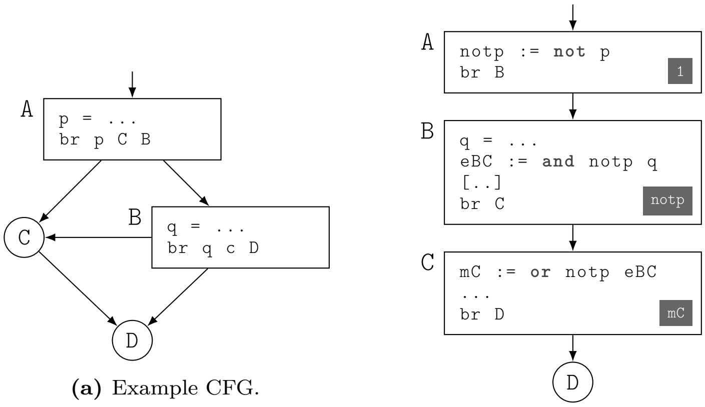
Figure 3.14.: Control-induced divergence and partial linearization.

Control-Induced Value Divergence. Re-consider the CFG in Figure 3.14b. The `x =` ϕ node is *varying* in spite of the fact that all incoming values are clearly *uniform* as they are constants. The reason for this is that the uniformity of `x` implicitly depends on the uniformity of `q`. This is an example for control-induced divergence: divergence in a branch causes divergence in a ϕ node. This relation becomes explicit in the partially linearized CFG in Figure 3.14c. There, the ϕ node has been converted to a `select` node that has `q` as a direct operand.

In Section 3.3, the vectorizer simply bailed upon divergent control flow. We now allow divergent branches, which brings control-induced divergence with it. The current divergence analysis will misclassify ϕ nodes as uniform as it is unaware of control-induced divergence.

Control-Divergence Analysis. We accompany the divergence analysis with a dedicated control-divergence analysis. Given a branch, the control-divergence analysis returns those blocks whose ϕ nodes implicitly depend on the branch condition[1]. That is, if-conversion would turn the ϕ nodes into `select` instructions that are data-dependent on the (former) branch condition. For the example in Figure 3.14b, the control-divergence analysis would compute that if `B` has a divergent branch, then all ϕ nodes in `D` are varying.

There exist various attempts in the literature to compute this dependence either directly or to augment the program such that this dependence could be inferred. As we show in Chapter 7, these approaches are often imprecise or even unsound. We present a novel control-divergence analysis that is optimal on DAGs and more precise than any other technique we could find on reducible CFGs.



(a) Example CFG.

(b) *Total* instructions (indicated by ":=" assignments).

```
A:
    p = ...
    // notp = not_v2(p,1)
    notp = not_v2(p)

B:
    q = ...
    // eBC = and_v2(notp, q, notp)
    eBC = and_v2(notp, q)

C: [..]
    // mC = or_v2(eBC, q, notp, mC)
    mC = or_v2(eBC, q)

    [..]
```

(c) Widening emits unpredicated SIMD code for *total* instructions. (Invalid) code without *total* in comments.

Figure 3.15.: Use of total instructions to compute block predicates.

Predicated Widening and Total Instructions. With the introduction of predicates into the program, we need to consider them in the generated SIMD code. We modify the widening phase as follows: If an instruction is free of side effects and *uniform*, widening keeps it scalar, ignoring the predicate. Otherwise, we widen the instruction by translating it to a predicated SIMD instruction (Section 2.7).

The result of predicated widening of the if-converted CFG in Figure 3.13b is shown in Figure 3.12b. For example, the load `v = B[i]` in Figure 3.12a is guarded by a divergent branch (the if statement). A load has potential side effects (accessing invalid memory may abort the program with an exception), hence, the load is widened to a predicated SIMD load (`masked_load_v4`).

Total Instructions. Note that instructions that compute the predicate cannot themselves be predicated. Else, the predicate-defining instruction would yield $\top$ (*undefined value*) on all lanes that execute it only passively. To this end, we introduce the concept of *total* instructions. Widening always emits unpredicated SIMD code for *total* instructions - a total instruction computes totally on all lanes, ignoring any block predicate.

Consider Figure 3.15a, a simple CFG with two divergent branches. Next to it, in Figure 3.15b, we show the if-converted CFG. Additional instructions have been inserted to compute the predicates of blocks `B` and `C`. The instructions defining `mC`, `eBC` and `notp` are all *total*. Therefore, widening emits the intended predication instructions into the SIMD code, shown in Figure 3.15c.

[1] This is also known as sync dependence [Coutinho et al., 2011].

Vectorizer Pipeline

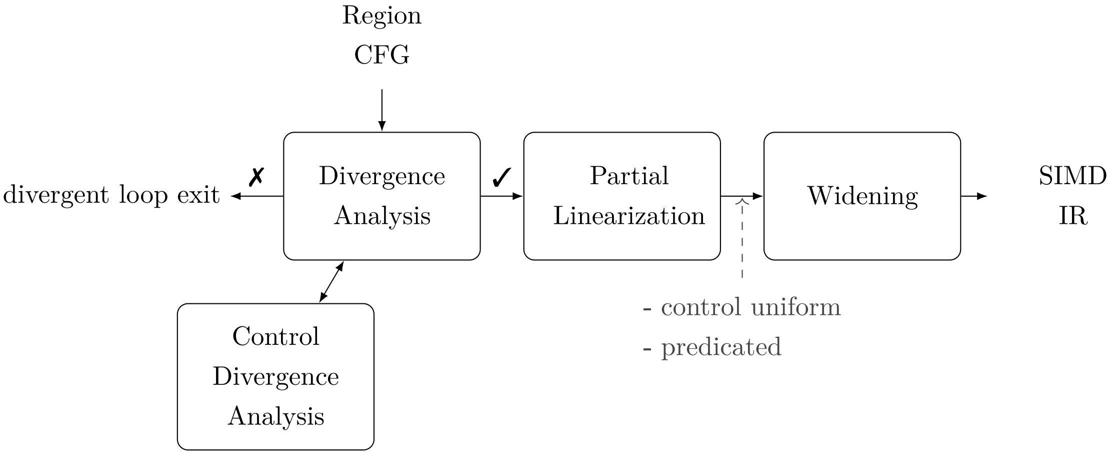


Figure 3.16.: Parallel loop vectorizer with partial linearization.

Figure 3.16 shows the extended vectorizer pipeline. The divergence analysis calls into the control-divergence analysis to detect value divergence upon encountering a divergent branch. Partial linearization runs after the divergence analysis, if-converting all *divergent* branches, and establishes uniformity of control flow. By virtue of *total* instructions, the predication logic is simply part of the program. Loops with divergent loop exit conditions still fail to vectorize.

3.5. Lock-step and Horizontal Operations

The vectorizer pipeline up to this point makes use of vector loads and stores, regular SIMD instructions and (implicitly) broadcasts. In this section, we introduce the notion of horizontal operations, their applications and how they can be represented in the vectorizer.

```
 1 float xsqrtf_u35(float d) {
 2   float q = 1.0f;
 3
 4   d = d < 0 ? SLEEF_NANf : d;
 5
 6
 7   bool scaleUp= \
 8     d < 5.2939559203393770e-23f;
 9   if (scaleUp) {
10     d *= 1.8889465931478580e+22f;
11     q = 7.2759576141834260e-12f;
12   }
13
14
15
16
17
18   bool scaleDown= \
19     d>1.8446744073709552e+19f;
20   if (scaleDown) {
21     d *= 5.4210108624275220e-20f;
22     q = 4294967296.0f;
23   }
24
25   // Fast inverse square root
26   int y = floatToRawIntBits(d + 1e-45) >> 1;
27   float x = intBitsToFloat(0x5f375a86 - y);
28
29   x = x * (1.5f - 0.5f * d * x * x);
30   x = x * (1.5f - 0.5f * d * x * x);
31   x = x * (1.5f - 0.5f * d * x * x);
32   x = x * (1.5f - 0.5f * d * x * x);
33
34   return (d == INFINITYf)?
35             INFINITYf
36             : (x * d * q);
37 }
```

(a) Scalar source code.

```
 1 float2 xsqrtf_u35_v2(float2 d) {
 2   float2 q = 1.0f;
 3
 4   float2 dneg = fcmp_olt_v2(d, 0.0)
 5   d = select_v2(dneg, SLEEF_NANf, d);
 6
 7   bool2 scaleUp = \
 8       fcmp_olt_v2(d, 5.2939559203393770e-23f);
 9   // if (scaleUp) {
10     float2 dUp = \
11         fmul_v2(d, 1.8889465931478580e+22f);
12     float2 qUp = 7.2759576141834260e-12f;
13
14     d = select_v2(scaleUp, dUp, d);
15     q = select_v2(scaleUp, qUp, 1.0);
16   // }
17
18   bool2 scaleDown = \
19       fcmp_ogt_v2(d, 1.8446744073709552e+19f);
20   // if (scaleDown) {
21   float2 dDown = \
22       fmul_v2(d, 5.4210108624275220e-20f);
23   float2 qDown = 4294967296.0f;
24
25   d = select(scaleDown, dDown, d);
26   q = select(scaleDown, qDown, q);
27   // }
28
29   // [..] // aligns with Line 26 of Figure 3.17a.
30 }
```

(b) Generated SIMD code.

Figure 3.17.: Approximate `sqrtf` function (from the *SLEEF 3.2* vector math library, adapted for presentation).

Motivating Example. Figure 3.17a shows the scalar implementation of an approximate square-root function taken from the SLEEF vector math library [Shibata et al., 2019]. In terms of the function structure there are three sections. In the first two sections, we find rescaling code if the input is below (Line 7 to Line 12) or above (Line 18 to Line 23) a certain threshold. The last section starting in Line 26 till the end, performs the actual computation.

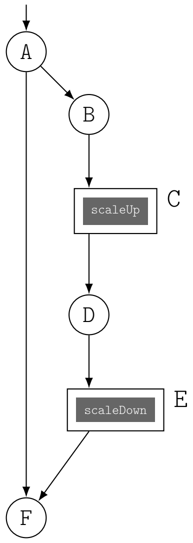
The whole-function vectorized version is shown in Figure 3.17b. In the scalar code, the rescaling code is only executed if the input is outside a certain range. Since rescaling is input dependent, the two cases are if-converted and will execute in any case. While the SIMD code is correct, it is rather inefficient if `xsqrt_u35_v2` is only called for inputs within the bounds for rescaling. If both conditions `scaleUp` and `scaleDown` evaluate to false for all inputs, `xsqrt_i35_v2` will perform a lot of operations that are unnecessary.

This is a situation where *horizontal operations* can be used to recover some of the control flow. In particular, we can make use of the horizontal *any* operation to skip the rescaling code if all inputs are within the range of no-rescaling. The operation `any(x)` evaluates to *true*, iff `x` is *true* for at least one thread. We call operations such as *any* horizontal because their result depends on the execution state across *all* threads. This is in contrast to vertical operations, such as `fadd`, whose per-thread result only depends on the individual state of each thread.

In the example of Figure 3.18, we manually insert *any* to test whether the branch conditions are *false* for all threads. The if statement in Line 13 guards the rescaling code in the way described above.

```
 1 float2
 2 xsqrtf_u35_v2(float2 d) {
 3 A:   float2 q = 1.0f;
 4
 5      dneg = fcmp_lt_v2(d, 0.0)
 6      d = select_v2(dneg, SLEEF_NANf, d);
 7
 8      bool2 scaleUp = \
 9          fcmp_olt_v2(d, 5.2939559203393770e-23f);
10      bool2 scaleDown = \
11          fcmp_ogt_v2(d, 1.8446744073709552e+19f);
12
13      if (any_v2(or_v2(scaleUp, scaleDown))) {
14 B:     // if (scaleUp) {
15 C:       float2 dUp = \
16              fmul_v2(d, 1.8889465931478580e+22f, scaleUp);
17          float2 qUp = 7.2759576141834260e-12f;
18        // }
19
20 D:     // if (scaleDown) {
21 E:       float2 dDown = \
22              fmul_v2(d, 5.4210108624275220e-20f, scaleDown);
23          float2 qDown = 4294967296.0f;
24        // }
25
26        dsU = select(scaleUp, dUp, d);
27        d = select(scaleDown, dDown, dsU);
28        qsU = select(scaleUp, qUp, 1.0);
29        q = select(scaleDown, qDown, qsU);
30      }
31
32 F: [..] // remainder as in Figure 3.17b
33 }
```

(a) SIMD code with *any* guard.



(b) CFG after partial linearization.

Figure 3.18.: Function `xsqrtf_u35` of Figure 3.17a vectorized with horizontal *any* test to skip over rescaling code.

The motivating example shows one important use case of horizontal SIMD instructions. They are useful to recover control flow in the presence of divergent branches. These branches can make the SIMD code more efficient and improve its runtime performance.

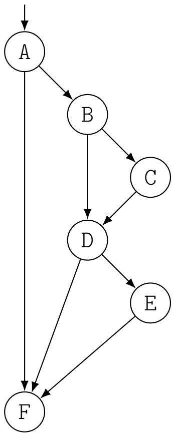
In earlier work such tests have been inserted *after* the widening stage [Shin, 2007]. However, there are good reasons to support horizontal operations throughout the vectorizer pipeline, starting with the initial region IR. The motivation for this is three-fold. First, there are data-parallel languages that support horizontal operations natively. For example, the ISPC language [Pharr and Mark, 2012] features the *coherent if* statement, which skips the a guarded code block if the predicate is all-false. By enabling *any* in the region IR, we enable an ISPC-like programming model with our vectorizer. Second, as shown in the example in Figure 3.19a, the *any* operation can be exposed to users as a builtin function. This enables users to insert coherent branches, as the one skipping the rescaling code, even in languages that are not originally data-parallel like ISPC. The SLEEF library, for example, is written in C. The `xsqrt` function with *any* guards can be compiled to LLVM IR and then auto-vectorized to match any SIMD ISA that has an LLVM backend. Third, even if *any* guards are inserted by the compiler, doing so before widening is a tractable solution, as we will show in Section 9.2. Early insertion of guards enables the reuse of subsequent transformations, such as partial linearization.

Lock-Step Execution. Up until now, we have looked at the region to vectorize as executing in parallel by multiple threads. However, the result of horizontal operators, such as *any*, depends on which threads execute it at the same time. Therefore, we define a more specific model of parallelism that makes precise this notion of threads executing an instruction together. This execution model is called lock-step execution.

In lock-step execution, all threads always execute the same instruction and at the same time. Due to branch divergence or predication, this may force threads into executing instructions they are not waiting to execute. Those threads only execute the instruction *passively*, i.e. the instruction executes without altering the execution state of the passive threads. Threads that are waiting for this instruction execute it *actively* and get the computed result and all side effects of the instruction.

```
1  bool any(bool v);
2
3  float xsqrtf_u35(float d) {
4  A:  float q = 1.0f;
5
6      d = d < 0 ? SLEEF_NANf : d;
7
8      bool scaleUp = \
9          d < 5.2939559203393770e-23f;
10     bool scaleDown = \
11         d > 1.8446744073709552e+19f;
12
13     if (any(scaleUp | scaleDown)) {
14 B:    if (scaleUp) {
15 C:      d *= 1.8889465931478580e+22f;
16         q = 7.2759576141834260e-12f;
17       }
18
19 D:    if (scaleDown) {
20 E:      d *= 5.4210108624275220e-20f;
21         q = 4294967296.0f;
22       }
23     }
24
25 F:  [..] // same as in Figure 3.17
26 }
```

(a)



(b)

t_0	t_1
A	A
B	B
D	D
	E
F	F

(c)

Figure 3.19.: Use of *any* in scalar code. 3.19b: CFG of Figure 3.19a. 3.19c: Execution with lock-step schedule.

Consider Figure 3.19, which shows an example for lock-step execution and how it assigns meaning to horizontal operators in scalar code. The code in Figure 3.19a is the scalar version of Figure 3.18a. It uses the *any* intrinsic in Line 13. Figure 3.19c shows an execution protocol for two threads executing Figure 3.19a in lock step. Each line of the protocol documents the execution of a basic block. If the block label shows in the column of a thread, it executes all instructions of that block actively. If there is an empty space, the thread still executes the same block as the other threads but does so passively.

In the beginning, both threads start at the entry block A and execute it actively. Assume that function parameter d has values such that scaleUp evaluates to 0 for both threads and that scaleDown evaluates to 0 for thread t_0 and to 1 for thread t_1. The code executes in lock step and thus both threads execute the *any* intrinsic in Line 13 together. Since scaleDown is 1 for thread t_1, the *any* intrinsic evaluates to 1 for both threads and both threads branch to block B. The threads proceed through B and skip C since the branch condition scaleUp is 0 for both threads. At the end of block D, thread t_1 branches to block E because scaleDown is 1 for that thread. The thread t_0 branches from D to F.

The next block to execute is E. We observe passive execution by thread t_0 as this thread is waiting to execute F. Thread t_1 actively executes block E and branches to block F next. Finally, both threads actively execute F.

We can read off from a lock-step execution protocol as the one in Figure 3.19c, the series of blocks as they are executing. We call this series, the *schedule* of the execution. In the example execution, the schedule is `A`, `B`, `D`, `E` and finally `F`. With lock-step execution and this schedule the *any* condition in Figure 3.19a behaves the same way as in the SIMD code in Figure 3.18a.

3.6. Divergent Loops and Semantic Interpretations

We have so far extended the vectorizer pipeline to handle divergent branches with partial linearization and predication. We introduced the concept of lock-step execution to the vectorizer pipeline to give horizontal operations a semantics before widening. The remaining obstacle to handling arbitrary divergent control flow is the problem of *divergent* loops. We initially referred to divergent loops as loops whose exit conditions depend on variables defined in the loop to vectorize. The trip count of a *divergent* loop depends on the thread. This is a problem for SIMD because threads executing in lock step may leave a divergent loop in different loop iterations.

```
 1 int
 2 logstar(double n) {
 3 A: x = n
 4    i = 0
 5 B: do {
 6      i += 1
 7      x = log(x)
 8
 9    } while (x > 1.0)
10 C: return select(n > 1.0, i, 0)
11 }
```

(a) Logstar function.

Block	$\mathtt{x}(\mathtt{t}_0)$	$\mathtt{x}(\mathtt{t}_1)$	$\mathtt{i}(\mathtt{t}_0)$	$\mathtt{i}(\mathtt{t}_1)$
A	88.38	1.5	0	0
B	4.48	0.41	1	1
B	1.15		2	
B	0.41		3	
C	0.41	0.41	3	1

(b) Example lock-step execution of Figure 3.20a.

Figure 3.20.

We show an instance of this in Figure 3.20a. The `logstar` function is given a `double` input and returns how often the `log` function can be applied to it before crossing below 1.0. We assume that `log(x)` silently returns *nan* when $\mathtt{x} \leq 0$. The loop exit condition of the loop in Line 9 depends on the thread, since every thread sees its own independent value for `n`.

```
int2
logstar_v2(double2 n) {
A: x = n
   int i = 0;
B: do {
     i += 1
     x = log(x)
     x_gt_one = fcmp_ogt_v2(x, 1.0)
   } while (any_v2(x_gt_one))
C: return select_v2(n > 1.0, i, 0)
}
```

(a) Naive SIMD translation (incorrect).

Block	$\mathtt{x}(\mathtt{t}_0)$	$\mathtt{x}(\mathtt{t}_1)$	$\mathtt{i}(\mathtt{t}_0)$	$\mathtt{i}(\mathtt{t}_1)$
A	88.38	1.5	0	0
B	4.48	0.41	1	1
B	1.15	nan	2	2
B	0.41	nan	3	3
C	0.41	nan	3	3

(b) Example execution of naive SIMD version in Figure 3.21a. The computed result for thread $\mathtt{t}_1$ is invalid.

Figure 3.21.: The naive SIMD translation of divergent loops is invalid.

If we execute the `logstar` function in lock step for two threads, we observe the execution as shown in the trace of Figure 3.20b. After the first iteration of the `while` loop, that is after the second line of the trace, the second thread leaves the loop. The first thread is still above 1.0 with `x` and keeps executing the loop two more times. Finally, both threads are at `C` when the first thread exits. The two threads compute the same correct result as if they had run the `logstar` function independently.

Loop Divergence. Intuitively, we would expect the SIMD code for the `logstar` function to look like the one shown in Figure 3.21a. Every scalar instruction is widened and we keep iterating the loop until all threads have left. However, this SIMD code is incorrect as we show with the trace in Figure 3.21b where we apply the `logstar_v2` to the same inputs as used in Figure 3.20b. After the first thread drops out (again second line), execution continues with B for both threads. The first thread computes the correct result. Yet, the second thread increments the variable i too often leading to an incorrect result. The result for the second thread is incorrect.

Transforming Divergent Loops. We need to transform divergent loops into uniform loops to be able to widen them to correct SIMD code. Section 9.1 presents a novel algorithm for transforming divergent loops that improves over the start of the art through its simplicity. While transforming divergent loops is not novel in itself, our algorithm is simpler than existing techniques because it can rely on partial control-flow linearization and the clearly defined lock step semantics of the P-LLVM IR, presented in this thesis. The divergent loop transform transforms divergent loops by addressing the two issues that arise with them: maintaining correct predicates for the loop blocks and keeping track of live-out values for threads that leave the loop before others.

We note that in Figure 3.21a all SIMD lanes are active in all loop iterations, regardless of how each thread evaluates the original loop exit condition x > 1.0. The solution to this is predication. We will switch to the CFG representation of logstar shown in Figure 3.22a. In Figure 3.22b, we added the predicate live for all instructions in the loop body. The predicate live disables threads that leave the loop early.

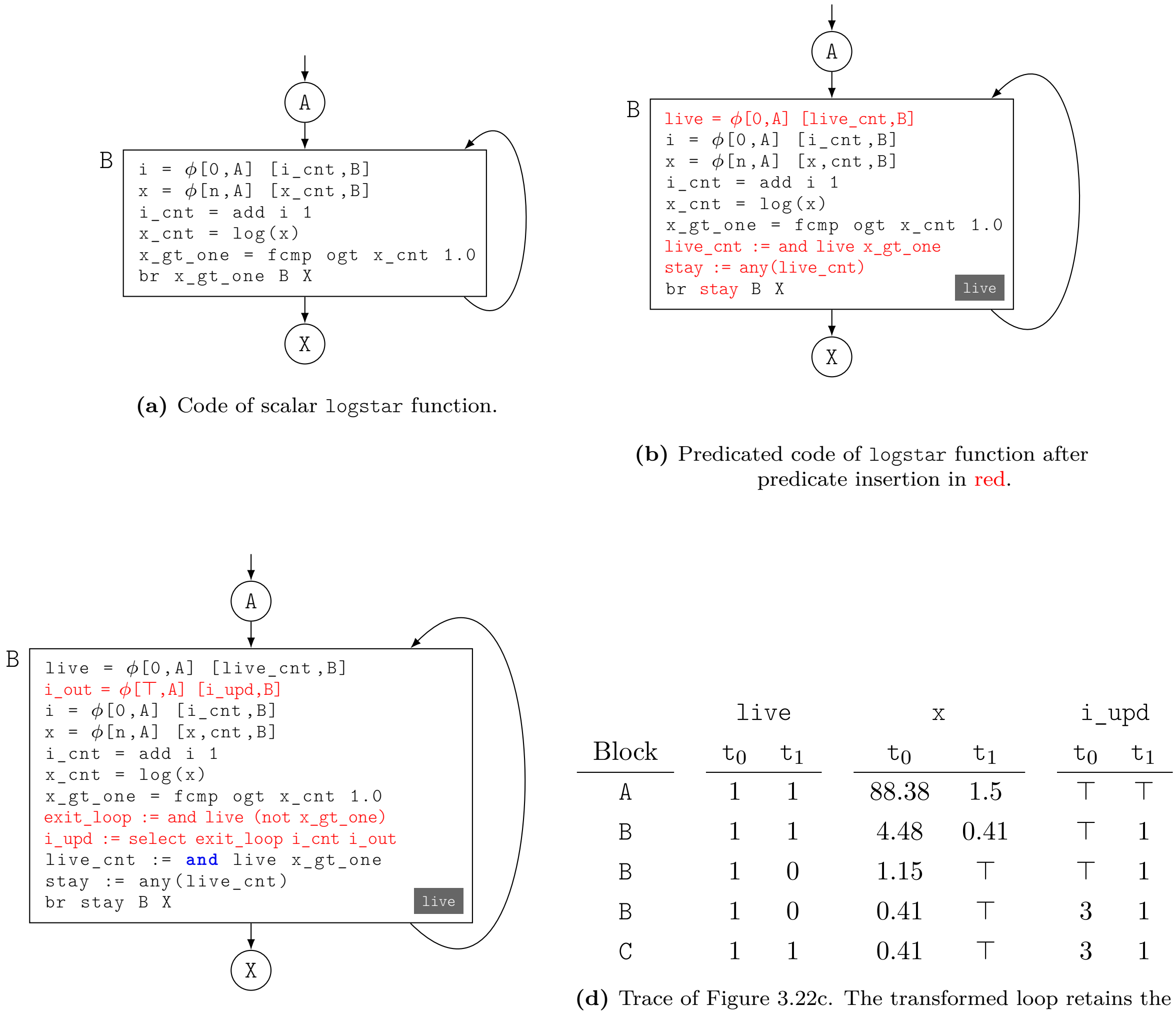
The second issue is that values of live-out variables have to be stored for each thread as it leaves the loop. These live-out values have to be retained even if the loop keeps iterating for other threads that still execute it. The required change is shown in Figure 3.22c. The variable exit_loop is true exactly when a thread is about to leave the loop. When that is the case i_upd will be assigned the current value of i_cnt. This value is then retained between the ϕ node i_out and i_upd. Figure 3.22d shows the trace for the transformed divergent loop. After each thread has left the loop, the value of i_upd holds the value of i at the time of exit, that is 3 for thread t_0 after the third execution of B ad 1 for t_1 after the first execution of B. Finally, when all threads have finished the loop, execution continues with X. All instructions outside the loop that used i_cnt before are modified to use i_upd instead, which holds the correct value of i_cnt as the thread left the loop.

3.6.1. Semantic Interpretations

Intuitively, we expect the code in Figure 3.22b to result in the same correct outcome as observed in Figure 3.20b. Since the exiting threads are masked-off with the predicate live the values of i and x should be preserved.

However, when we widened this code it will be similar to the code shown in Figure 3.21a, except for loop predication. Thus, when executing it for the same inputs, we obtain the same incorrect results as in the trace in Figure 3.21b. The values for x and i are set to $\top$ in the first iteration after a thread has left (line three for t_1). How can this happen despite the fact that B has a correct loop predicate? There is a mismatch between the intuitive semantics of lock-step and the semantics provided by SIMD. In the widened SIMD code, variables of threads that leave the loop early are overwritten. Yet, in the intuitive lock-step semantics, threads that exit the loop retain the values defined in their last loop iteration. Passive execution, caused by other threads that are still in the loop, will not overwrite them. This mismatch becomes an issue in divergent loops because only loops execute instructions repeatedly and with threads leaving the loop earl passive execution can follow active execution for the same instruction. Our solution to model this semantic gap is to define different semantic interpretations of the code, the FREEZING and OBLIVIOUS interpretations.

(FR)EEZING interpretation. We expect programs to behave as in Figure 3.20b. In the expected semantics, variables are unaffected by passive execution, i.e. no variable will change for those threads that are not waiting to execute the schedule instruction. We will call this the *FREEZING* (FR) interpretation



```
i = φ[0,A] [i_cnt,B]
x = φ[n,A] [x_cnt,B]
i_cnt = add i 1
x_cnt = log(x)
x_gt_one = fcmp ogt x_cnt 1.0
br x_gt_one B X
```

(a) Code of scalar `logstar` function.

```
live = φ[0,A] [live_cnt,B]
i = φ[0,A] [i_cnt,B]
x = φ[n,A] [x,cnt,B]
i_cnt = add i 1
x_cnt = log(x)
x_gt_one = fcmp ogt x_cnt 1.0
live_cnt := and live x_gt_one
stay := any(live_cnt)
br stay B X
```

(b) Predicated code of `logstar` function after predicate insertion in red.

```
live = φ[0,A] [live_cnt,B]
i_out = φ[⊤,A] [i_upd,B]
i = φ[0,A] [i_cnt,B]
x = φ[n,A] [x,cnt,B]
i_cnt = add i 1
x_cnt = log(x)
x_gt_one = fcmp ogt x_cnt 1.0
exit_loop := and live (not x_gt_one)
i_upd := select exit_loop i_cnt i_out
live_cnt := and live x_gt_one
stay := any(live_cnt)
br stay B X
```

	live		x		i_upd	
Block	t_0	t_1	t_0	t_1	t_0	t_1
A	1	1	88.38	1.5	⊤	⊤
B	1	1	4.48	0.41	⊤	1
B	1	0	1.15	⊤	⊤	1
B	1	0	0.41	⊤	3	1
C	1	1	0.41	⊤	3	1

(d) Trace of Figure 3.22c. The transformed loop retains the live out value of `i` in `i_upd`.

(c) `i_upd` and `i_out` record the live out value of `i`. Instructions outside the loop use `i_upd` instead of `i`. This widens to correct SIMD code.

Figure 3.22.: Transforming a divergent loop. Added instructions colored red.

of lock step. When a thread drops out of a divergent loop and there is predication, we expect that `i` and `x` retain their old values. The variables are *frozen* and will not change when their assigning instruction is masked-off.

The FREEZING semantics is useful because it models the expected lock-step execution of divergent loops.

(OBL)IVIOUS interpretation. However, the generated SIMD code breaks with the expected, lock-step semantics and behaves as shown in Figure 3.21b. This is because predicated SIMD instructions are defined to return ⊤ on passive lanes. In a way, the old values of the variables are "forgotten" by masked-off execution. We lift this behavior to code *before* widening by introducing the concept of

*OBL*IVIOUS (`OBL`) semantics. Under the *OBLIVIOUS* (`OBL`) interpretation, instructions that execute masked-off set their variable to $\top$.

The OBLIVIOUS semantics is useful because it models the behavior of widened SIMD code, before any SIMD code is generated.

With the two semantic interpretations, FREEZING and OBLIVIOUS, we can consider the issue of transforming divergent loops from a new angle. The *Divergent Loop Transform* converts a code that is correct under FREEZING semantics into a code that is correct under OBLIVIOUS semantics. If code is correct under OBLIVIOUS semantics, we know that widening will generate correct SIMD code. We present the *Divergent Loop Transform* in Section 9.1.

3.7. RV Vectorization System

We extend the vectorizer to deal with the effects of divergent loops as shown in Figure 3.23. This brings about two changes to the vectorizer pipeline: First, we introduce two semantic interpretations, the FREEZING interpretation and the OBLIVIOUS interpretation. The FREEZING interpretation holds at the beginning of the vectorizer pipeline as it models the intuitive semantics of divergent loops: Threads that exit a divergent loop keep the variables as they were in their last iteration. The OBLIVIOUS interpretation models the semantics of the SIMD code at the end of the pipeline: If a thread leaves a divergent loop before other threads its variables are re-assigned even under passive execution. Second, we insert the divergent loop transform, which transforms divergent loops into uniform loops. If all loops are uniform then the code is correct under both semantic interpretations, Therefore, the pipeline proceeds with partial linearization and finally the widening phase emits SIMD code.

Figure 3.23 shows the full vectorizer pipeline of the *Region Vectorizer (RV)*, which implements the approach developed in this thesis.

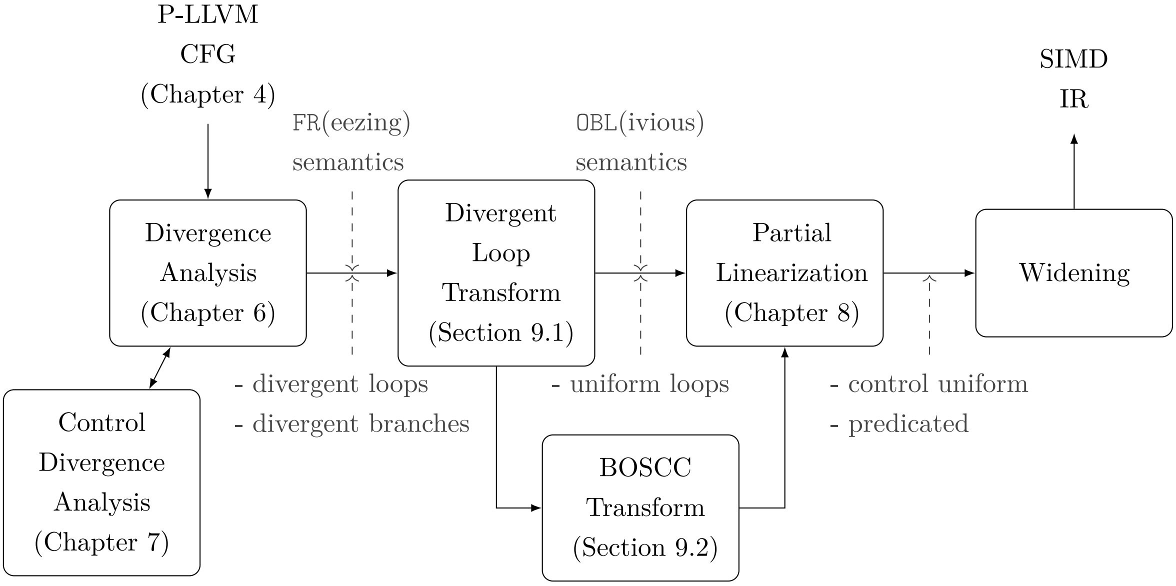


Figure 3.23.: RV Vectorizer Pipeline.

The chapters referenced in Figure 3.23 provide a complete technical description of the components, including related work.

The P-LLVM intermediate representation augments LLVM IR with lock-step semantics, predicates and horizontal operations.

The *BOSCC Transform* inserts branches that skip over code with an *all-false* predicate. Section 9.2 demonstrates how this classic optimization has a simple implementation, exploiting the guarantees of P-LLVM and *Partial Linearization.*

The Divergence Analysis outlined in this chapter only distinguishes between *uniform* and *varying* variables. These two states are the elements of a basic divergence-analysis lattice, the abstract lattice used by the divergence analysis. In Chapter 10, we extend this basic lattice to the Stride-Alignment lattice (sa-lattice). The refined sa-lattice enables the divergence analysis to detect *uniform* branches in more cases and aides the widening stage to generate faster SIMD code.

The description of the RV system is complete after Chapter 10.

TensorRV. The notions of thread array and SIMD vectors are inherently one-dimensional. Chapter 11 extends the *RV* system to *TensorRV*, which generalizes one-dimensional vectorization to multidimensional tensorization.

Chapter 4.

P-LLVM: Data-parallel, Predicated IR

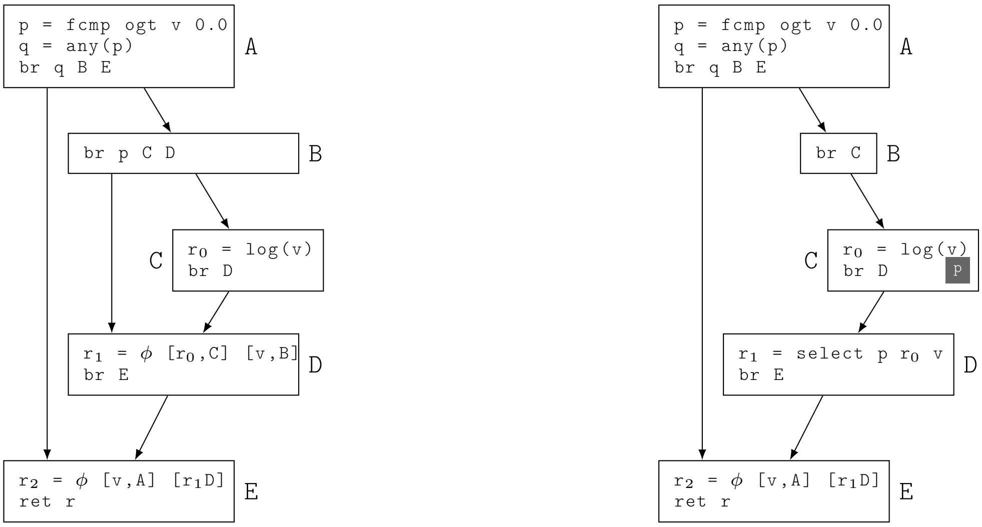
P-LLVM is a novel, lock-step extension of LLVM IR. P-LLVM is the central program representation of the RV vectorization system. The P-LLVM language is a light-weight extension of LLVM IR: it adds predicates to basic blocks and two new keywords to control the behavior of passive threads[1]. This chapter introduces P-LLVM and its formal semantics.

```
double
safelog(double v) {
A: double r = v;
   bool p = v > 0.0;
   if (any(p)) {
B:   if (p) {
C:     r = log(v);
D:   }
E: }
   return r;
}
```

(a) Data-parallel program with horizontal *any* intrinsic.

```
double
safelog_v4(double4 v) {
   double4 r = v;
   bool4 p = v > 0.0;
   q = any_v4(p)
   if (q) {
     t = log_v4(v,p);
     r = select(p,t,r)
   }
   return r;
}
```

(b) SIMD code widened from Figure 4.1d.



(c) P-LLVM representation of Figure 4.1a. *any* has a precise semantics (Section 4.9.2).

(d) Partially linearized. The mask of C (gray box) is part of the IR (Section 4.4).

Figure 4.1.: The RV vectorization system uses P-LLVM as its central program representation from the frontend (4.1a) to SIMD code generation by widening (4.1b).

[1] Threads execute a block passively if the block's predicate is 0 or the block is not the one that the thread is waiting to execute.

P-LLVM accomplishes three goals:

1. Data-parallel programs given in regular LLVM IR are already P-LLVM programs. Any compiler frontend for LLVM that requires data-parallel vectorization can immediately leverage the P-LLVM language and the RV vectorization system that is built on it. Data-parallel LLVM IR programs originate from diverse sources, such as data-parallel languages (OpenCL, SPIR-V) or parallel loops. Figure 4.1c shows the P-LLVM representation of the function Figure 4.1a that is whole-function vectorized (Section 3.3). The *any* intrinsic[2] is a horizontal operator a common feature of data-parallel languages and directly representable in P-LLVM.

2. With P-LLVM as its basis, vectorization becomes a pipelined process of successive program transformation. The program finds in P-LLVM a well-defined semantics at every stage of the vectorization pipeline. This is a significant improvement over existing vectorization systems and data-parallel compiler IRs. Prior data-parallel compiler IRs are not as expressive as P-LLVM, proprietary or not as rigorously defined. Earlier vectorization systems either treat the IR as a code artifact whose semantics depends on the exact stage in the vectorization pipeline (e.g. WFV) or the systems explicitly switch program representations (e.g. PDG to CFG). P-LLVM stands out for its combination of expressiveness, suitability as a vectorizer IR and semantical rigor. Figure 4.1d shows the program after partial linearization (Chapter 8) has removed the divergent branch in CFG in Figure 4.1c. The block predicate of `C` is part of the P-LLVM prorgram.

3. P-LLVM programs that are free of divergent branches, that is they are control uniform, transparently translate into predicated SIMD code: Every P-LLVM instruction is widened into one SIMD instruction. All P-LLVM programs that have control-divergent branches or loops can be brought into control uniform form before widening takes place. In prior systems, such as ISPC and WFV, widening is much more complex and non-local: e.g. in the worst case of a divergent loop live-out, the extra SIMD code for just one instruction may be several blend instructions spread out to the loop header, latch and exit blocks. Figure 4.1b shows the SIMD code generated by widening the instructions in Figure 4.1d.

This chapter is conceptually divided into three parts. In part one (Section 4.1 to Section 4.4), we define the syntax and execution state of a P-LLVM program, the runtime schedule that drives its execution and the concept of thread activation. The second part (Section 4.5 to Section 4.10) presents the inference rules that define the formal small-step semantics of P-LLVM programs. The last section of this part establishes well-formedness criteria for P-LLVM programs. The final part (Section 4.11), places P-LLVM in the context of related work.

[2] $any(p)$ returns true for all threads if p is true for at least one thread.

4.1. The P-LLVM Language

Program	prg	::=	$\overline{b}$
Blocks	b	::=	$\ell\ val_{msk}\ \overline{phi}\ \overline{c}\ tmn$
Constants	$cnst$	::=	$\textbf{int}N$ N-bit integer literal $\mid$ $\textbf{fp}N$ N-bit floating point literal
Types	typ	::=	$\textbf{int}N \mid \textbf{fp}N \mid \ldots$
Values	val	::=	$id \mid cnst$
Call	$callop$	::=	$fnc\,(\,\overline{val})$
Offset	$gepop$	::=	$\textbf{elemptr}\ val_{ptr}\ \overline{val}$
Terops	$terop$	::=	$\textbf{select} \mid \ldots$
Binops	bop	::=	$\textbf{add} \mid .. \mid \textbf{fcmp}\ P_{fp} \mid .. \mid \textbf{store} \mid \ldots$
Unops	$unop$	::=	$\textbf{not} \mid \textbf{load}\ typ \mid \ldots$
Instructions	$inst$	::=	$terop\ val_0\ val_1\ val_2 \mid bop\ val_0\ val_1 \mid unop\ val_0 \mid$
			$\textbf{alloca}\ typ \mid gepop \mid callop$
Commands	c	::=	$id = inst \mid id = inst\ \textbf{total} \mid id = \textbf{mask}()$
Phis	phi	::=	$id = \phi\ typ\ \overline{[val_j, \ell_j]}$
			$id = \phi\ typ\ \overline{[val_j, \ell_j]}\ \textbf{shadow}(val_s)$
Terminators	tmn	::=	$br\ \ell \mid br\ val_{cnd}\ \ell_0\ \ell_1 \mid ret\ typ\ val_{res}$
Incoming Block	ℓ_{in}	::=	$\ell \mid \perp$
Next Block	ℓ_{next}	::=	$\ell \mid \top \mid \Box(r \in \mathcal{V})$
Successor Index	i	::=	$\mathbb{N} \mid \Box(r \in \mathcal{V}) \mid \top$

Figure 4.2.: P-LLVM Language Reference. Data-parallel language features highlighted in blue. Productions in *italics* are extended syntax used in the formalization.

The P-LLVM language is an extension of the LLVM intermediate representation, discussed in Section 2.1. Figure 4.2 shows the syntax of the P-LLVM IR. The formalization and grammar of P-LLVM is inspired by VE-LLVM [Zhao et al., 2012]. We provide a formal grammar to use it in the forthcoming formalization of the semantics of P-LLVM. The actual implementation uses standard LLVM IR enriched with light-weight annotations and additional builtin functions. Definition 1 defines the components that make up a P-LLVM program.

Definition 1. *(P-LLVM Program) A P-LLVM program consists of:*

- *A list of basic blocks (prg). The first block of the list is the entry block of the program. Each basic block consists of a unique label (ℓ), a predicate value (val_{msk}), a list of ϕ nodes ($\overline{phi}$), a list of commands ($\overline{c}$) and ends in a terminator (tmn).*
- *A function map $\theta : fnc \to \big((\mathcal{V}^W)^* \times Bit^W \times \mathbb{M} \to \mathcal{V}^W \times \mathbb{M}\big)$. The function θ maps function symbols (fnc) to mathematical functions. It maps to functions that take a list of vector arguments, an activation vector and a memory state. The functions return a vector of result values and the resulting memory state.*

The ϕ nodes, instructions and the terminator of a block execute under the influence of the block predicate (val_{msk}). In the following figures featuring P-LLVM programs, we depict the block predicate in a gray box in the lower right corner of the block. For example in Figure 4.6a, `notp` drawn into the block `C` denotes that the val_{msk} of block `C` is the given by the variable `notp`. Where omitted, the block predicate is given by the constant *true*, the block is effectively unpredicated. Data-parallel LLVM IR programs translate into P-LLVM programs by assigning to each block the constant *true* predicate.

For the sake of simplicity, the functions names (*fnc*) used in function calls (*callop*) are defined as mathematical functions provided by the function map θ. For example, θ contains common mathematical functions: $\theta(sin)$ is the vectorized sine function. We discuss the function map in more detail in Section 4.9.2.

A P-LLVM program induces a Control-Flow Graph in the following way. The block labels of P-LLVM program (ℓ) make up the nodes of the CFG (set `V`). The successor labels of a terminator (*tmn*) define the control-flow edges (`E`) out of the block containing the terminator. The entry block of the CFG (*entry*) is the entry block of the P-LLVM program (first block of *prg*).

Data-parallel Modifiers. P-LLVM extends the syntax of regular LLVM IR with two modifiers and one additional instruction, highlighted in blue in Figure 4.2. The motivation for this is to give instructions and ϕ nodes a semantics under passive execution. Passive execution arises when the predicate of a block evaluates to false or in the presence of divergent control flow. The ϕ nodes have an optional operand (**shadow**). A ϕ node with a shadow operand resolves to the shadow input under passive execution. An instruction with a **total** modifier is not subject to passive execution. Finally, the special instruction **mask** returns the activation of the basic block.

We provide a formal small-step semantics [Plotkin, 2004] for the aspects of P-LLVM that differ from regular LLVM IR.

4.2. Execution State

P-LLVM programs are executed in data-parallel fashion by W threads. Each thread is uniquely named by its thread identifier $\mathtt{t} \in \mathcal{T}$ as of Definition 2.

Definition 2. *(Thread Array) A P-LLVM program is executed by an array of $W \in \mathbb{N}$ many threads where each thread has a unique identifier in $\mathcal{T} = \{ 0, 1, .., W - 1 \}$.*

The purpose of the P-LLVM intermediate representation is vectorization. In the process, the data-parallel threads are mapped to the lanes of SIMD operations and variables. That is the lane at position $\mathtt{t}$ in the generated SIMD code will realize the corresponding operation of thread $\mathtt{t}$ in the P-LLVM program.

All state of P-LLVM execution is captured in the P-LLVM execution state tuple as of Definition 3.

Definition 3. *(P-LLVM Execution State) The execution state of a P-LLVM program for W many threads is given by a tuple $\sigma = \left(\mathtt{M}, \Delta^W, {\ell_{in}}^W, {\ell_{next}}^W\right)$. Let $\mathtt{t} \in \mathcal{T}$ be the unique identifier of a thread, then*

- *$\mathtt{M} \in \mathbb{M}$ is the memory state, which is shared among all threads. This part of the state is read and written by load, store and call instructions. $\mathbb{M}$ is the set of all possible memory states.*
- *$\Delta^W(\mathtt{t})$ is the private mapping from variables to values for thread $\mathtt{t}$.*
- *The incoming block ${\ell_{in}}^W(\mathtt{t})$ indicates the block that the thread executed last. It is either a block label or the special symbol $\bot$ to indicate that no predecessor was executed.*
- *The next block ${\ell_{next}}^W(\mathtt{t})$ is the block that the thread executes next. It is either a block label (the branch target), the special token $\square(r)$ to indicate that the thread has terminated with return value $r \in \mathcal{V}$, or the special token $\top$ to signal that the thread will accept any successor of its incoming block as branch target.*
- *We call the pair of incoming block ${\ell_{in}}^W(\mathtt{t})$ and next block ${\ell_{next}}^W(\mathtt{t})$ the* control state *of thread $\mathtt{t}$.*

Since W many threads execute a P-LLVM program in parallel, the parts of the execution state that are private to each thread exist in W many copies. For example, each thread has a private mapping from variables to values and so the variable mapping in the execution state is given by Δ^W.

The basic block labels ${\ell_{in}}^W$ and ${\ell_{next}}^W$ model the threads' control state. Conceptually, the control state characterizes a thread as standing on an edge of the CFG: The *next block* ${\ell_{next}}^W(\mathtt{t})$ is either the branch target block or the special token $\square$ to signal that a return statement was reached. The *incoming block* ${\ell_{in}}^W(\mathtt{t})$ is the block that a thread has last executed or the special token $\bot$ to indicate that there was no predecessor this thread is reaching from.

Just as in sequential execution of a regular LLVM IR program, the execution of a P-LLVM program begins at the entry block. Initially, $\ell_{next} = next$ and since no predecessor was executed $\ell_{in} = \bot$. This is formalized by Definition 4.

Definition 4. *(Initial State) A P-LLVM execution state $\sigma_{init} = \left(\mathtt{M}, \Delta^W, {\ell_{in}}^W, {\ell_{next}}^W\right)$ is called an* initial state *of a P-LLVM program, if for all $\mathtt{t} \in \mathcal{T} . {\ell_{next}}^W(\mathtt{t}) = entry$ and ${\ell_{in}}^W(\mathtt{t}) = \bot$.*

When finally a thread executes a return statement, the thread sets $\ell_{next} = \square(r)$ where the symbol $\square$ means that the thread has entered a terminated state and $r \in \mathcal{V}$ is the return value. For sake of brevity, we frequently omit the returned value where it is inconsequential, stating just $\ell_{next} = \square$.

Throughout this chapter, we take the liberty to assume the perspective of a single thread while defining and explaining concepts of P-LLVM. We do so by omitting the W superscript of formal functions and variables. For example, ${\ell_{next}}^W$ is simply the W tuple as defined in the execution state. Referring to ℓ_{next} however, we imply that the statement applies to any thread $\mathtt{t} \in \mathcal{T}$. This is in contrast to the explicit quantification for all threads, e.g. as used in the wording of Definition 4.

4.3. Runtime Schedule and Control Mask

Despite their private control state, threads do not execute the program independently. The execution is coordinated by a scheduler that is not part of the threads' execution state. The scheduler produces a block label and all threads have to execute that block. This continues until all threads have executed a return statement and entered into a terminated state, i.e. $\ell_{next} = \square$. For the purposes of P-LLVM semantics, the only thing that matters about the scheduler is its decisions, the scheduled blocks, and not how it makes those decisions. Therefore, the formalization represents the scheduler through the *runtime schedule*, the list of block labels in the order they are scheduled. This runtime schedule is formalized in Definition 5.

Definition 5. *(Runtime Schedule) A runtime schedule is a list of basic blocks $\overline{\ell_{sc}} \in \mathtt{V}^*$. We call $\overline{\ell_{sc}}$ an* initial *schedule, if $\overline{\ell_{sc}}(0) = entry$. Likewise, we call $\overline{\ell_{sc}}$ a* terminating *schedule for a given P-LLVM state, if executing the schedule leads all threads to a terminated control state. That is when the runtime schedule is completely executed it holds in the resulting execution state that $\forall \mathtt{t} \in \mathcal{T}.\ell_{next}(\mathtt{t}) = \square$. Finally, a schedule is called* complete *if it is initial and terminating.*

We stress that runtime schedules are entirely independent of the execution state: for any execution state of any P-LLVM program any sequence of basic blocks of that program is a valid runtime schedule. We emphasize this aspect of P-LLVM because it breaks with a standard staple in data-parallel languages: it is common for language semantics to derive scheduling decisions from the execution state, e.g. by defining a stack-based scheduling mechanism that takes the threads' branching decisions into account [Farrell and Kieronska, 1996; Leißa, 2017; Habermaier and Knapp, 2012; Karrenberg, 2015; Coutinho et al., 2011; Sampaio et al., 2013; Alur et al., 2017]. We discuss restrictions on the schedule space in Chapter 5. However, those restrictions are made by conditioning the schedule space a-posteriori, i.e. not as part of language semantics and different schedule regimes are thinkable.

In P-LLVM threads always execute in lock step: all threads concurrently execute the same instruction at the same time. When a block is scheduled, the threads jointly execute the instructions in that block in lock step. This means that the control state of a thread determines how (but not if) it executes the instructions of a scheduled block. Threads execute each scheduled block in one of two states, *active* or *passive*, which we refer to as the activation of the thread. Intuitively, under active execution a thread performs the operations in the scheduled block while under passive execution it does not. If a thread's control state disagrees with the scheduled block then it will only execute it passively. Otherwise, if the thread's control state agrees with the scheduled block and the predicate of the block evaluates to true, then the thread will execute the block actively.

The control state aspect of the activation is given by a thread's control mask (*cmsk*), which is formalized by Definition 6.

Definition 6. *(Control Mask)*

$$cmsk\ \ell_{in}\ \ell_{next}\ \ell_{sc} = \begin{cases} 1 & \textit{if } \ell_{next} = \top \quad (\ell_{in}, i, \ell_{sc}) \in \mathtt{E} \quad i \in \mathbb{N} \\ 1 & \textit{if } \ell_{next} = \ell_{sc} \\ 0 & \textit{otherwise} \end{cases}$$

The second and third case of Definition 6 cover the intuitive outcomes of the control mask. In the second case, the scheduled block is the next block of the thread. In the third case, it is not and the thread executes passively. Threads that have executed a return statement will remain in this case and never attain an active control mask again. In the remaining first case, it holds that $\ell_{next}(\mathtt{t}) = \top$; The thread t will actively execute any of its successors as soon as it is scheduled. The control mask of the entire thread array is given by $\mathit{cmsk}^W = \mathit{map}\ \mathit{cmsk}$, e.g. the bit vector resulting from applying *cmsk* to each individual thread.

Schedule Semantics. The *schedule transition*, defined by [LSS-W-STEP], formalizes the effect of consuming the head of the runtime schedule and executing it for all basic blocks.

$$\frac{\begin{array}{c}\ell_{sc} \vdash \left(\mathtt{M}, \Delta^W, {\ell_{in}}^W, {\ell_{next}}^W\right) \rightarrow_{\mathcal{I}} \mathtt{M}'\ \Delta'^W\ i^W \\ \mathit{advancepc}^W\ i^W\ {\ell_{in}}^W\ {\ell_{next}}^W\ \ell_{sc} \rightarrow {\ell_{in}}'^W\ {\ell_{next}}'^W\end{array}}{\ell_{sc}\overline{\ell'_{sc}}\ \left(\mathtt{M}, \Delta^W, {\ell_{in}}^W, {\ell_{next}}^W\right) \xrightarrow{\ell_{sc}}\!\!\!\!\twoheadrightarrow_{\mathcal{I}} \overline{\ell'_{sc}}\ \left(\mathtt{M}', \Delta'^W, {\ell_{in}}'^W, {\ell_{next}}'^W\right)}\ \text{[LSS-W-STEP]}$$

Figure 4.3.: State transition for executing the next block on the schedule.

We say that a block is *scheduled* meaning that the block is the head of the schedule and a transition of [LSS-W-STEP] takes place. The rule, shown in Figure 4.3, has two premises. The one on top is the basic block transition ([BLS-W-STEP] defined in Section 4.7), which models the execution of the content, i.e. the ϕ nodes, instructions and the terminator, of the scheduled basic block ℓ_{sc}. Executing the block results in an updated variable environment (Δ'^W) and memory state ($\mathtt{M}'$). Also, it yields a successor index vector (i^W), which contains the branching decisions of the threads. The actual control state update is performed in the second premise: *advancepc* takes the just computed successor index, the control state and the scheduled block into account. The exact definition of *advancepc* is shown in Figure 4.4.

$$\boxed{\mathit{advancepc}\ i\ \ell_{in}\ \ell_{next}\ \ell \rightarrow \ell\ \ell_{next}}$$

$$\frac{\mathit{cmsk}\ \ell_{in}\ \ell_{next}\ \ell_{sc} \qquad i \in \mathbb{N} \qquad \left(\ell_{sc}, i, {\ell_{next}}'\right) \in \mathtt{E}}{\mathit{advancepc}\ i\ \ell_{in}\ \ell_{next}\ \ell_{sc} \rightarrow \ell_{sc}\ {\ell_{next}}'}\ \text{[PC-ACTIVE]}$$

$$\frac{\mathit{cmsk}\ \ell_{in}\ \ell_{next}\ \ell_{sc} \qquad i = \top}{\mathit{advancepc}\ i\ \ell_{in}\ \ell_{next}\ \ell_{sc} \rightarrow \ell_{sc}\ \top}\ \text{[PC-ND-BRANCH]}$$

$$\frac{\neg \mathit{cmsk}\ \ell_{in}\ \ell_{next}\ \ell_{sc}}{\mathit{advancepc}\ i\ \ell_{in}\ \ell_{next}\ \ell_{sc} \rightarrow \ell_{in}\ \ell_{next}}\ \text{[PC-PASSIVE]}$$

$$\frac{\mathit{cmsk}\ \ell_{in}\ \ell_{next}\ \ell_{sc} \qquad i = \square(r)}{\mathit{advancepc}\ i\ \ell_{in}\ \ell_{next}\ \ell_{sc} \rightarrow \ell_{sc}\ \square(r)}\ \text{[PC-TERM]}$$

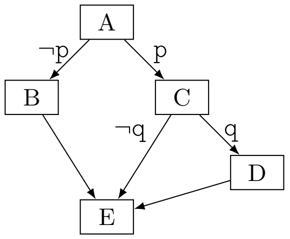
$$\frac{\forall \mathtt{t} \in \mathcal{T}.\ \left(\mathit{advancepc}\ i^W(\mathtt{t})\ {\ell_{in}}^W(\mathtt{t})\ {\ell_{next}}^W(\mathtt{t})\ \ell_{sc} \rightarrow {\ell_{in}}'^W(\mathtt{t})\ {\ell_{next}}'^W(\mathtt{t})\right)}{\mathit{advancepc}^W\ i^W\ {\ell_{in}}^W\ {\ell_{next}}^W\ \ell_{sc} \rightarrow {\ell_{in}}'^W\ {\ell_{next}}'^W}\ \text{[PC-WSTEP]}$$

Figure 4.4.: Control state update after a scheduled block has been executed.

The rules for *advancepc* break down into two sets depending on whether the control mask of the thread is active (rules [PC-ACTIVE] and [PC-ND-BRANCH] and [PC-TERM]) or not ([PC-PASSIVE]). In the latter case, the control state is unchanged. In case of an active control mask, the terminator either yields a successor index ($i \in \mathbb{N}$), a termination signal with return value ($ret(r \in \mathcal{V})$) or $\top$. In the integer index case control simply advances to that successor in the CFG ([PC-ACTIVE]). If the successor index is $\top$ the thread advances to *any* successor of it its last executed block as soon as such a block is scheduled.

A terminated thread stays terminated through [PC-PASSIVE] for the special case $\ell_{next} = \square$.

Example: Control Mechanics. We shown an example for the execution of a P-LLVM program with a runtime schedule in Figure 4.5. Figure 4.5a shows the CFG of the program we are executing. We turn our attention to Figure 4.5b, which shows the runtime schedule, the control state and control mask before each scheduled block executes, and the resulting branching decision (successor index). Both threads start at the entry block `A`. As shown in the branching decision column (i^W), the first thread branches to `B` at successor index 0 and the second thread to `C` at successor index 1. The next scheduled block is `B`. This is the next block of the first thread, and so after executing `B`, the first thread updates its control state to execute `E` next. The last executed block `B` is the incoming block. Looking at the second thread when `B` is scheduled, we observe that because `B` is not the next block of this thread (but `C`), the first thread executes only passively resulting in $\top$ as its branching decision. This does not affect the first thread's control state however, since the control mask for `B` was 0. When `C` is scheduled, we see a similar outcome as for `B` with the roles of the first and second thread swapped: the first thread executes passively, the second actively. In effect, the second thread branches to `D`, the first stalls. The same happens for `D`. Finally, both threads actively execute `E`. This is the next block of the first thread after `B` was scheduled and the second thread branches to `E` after `D`. Both threads reach the return statement at the end of `E` and set their next block to $\square$, signalling a terminated state.



(a) Example CFG. Branch conditions drawn onto edges.

ℓ_{sc}	${\ell_{in}}^W$		${\ell_{next}}^W$		$cmsk^W$		i^W	
A	$\bot$	$\bot$	A	A	1	1	0	1
B	A	A	B	C	1	0	0	$\top$
C	B	A	E	C	0	1	$\top$	1
D	B	C	E	D	0	1	$\top$	0
E	B	D	E	E	1	1	$\square$	$\square$

(b) Execution with runtime schedule.

Figure 4.5.: 4.5a: CFG of a P-LLVM program. 4.5b: Execution protocol for two threads with the runtime schedule of the firs column. Second, third columns: Control state before executing the block. Fourth column: Control mask. Last column: Branching decision (successor index) after the block was executed.

4.4. Predication and Activation

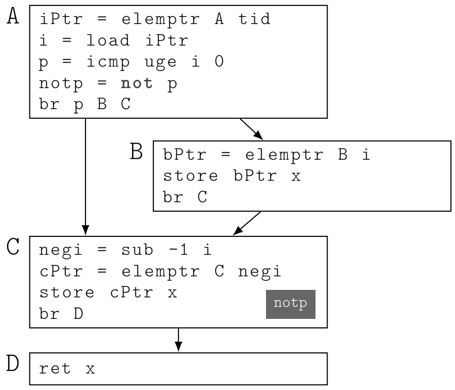
Besides the control mask, active execution is also controlled by the evaluation result of the block predicate, defined by Definition 7.

Definition 7. *(Predicate Mask) The predicate mask (pmsk) of a thread is the value of the block predicate of the scheduled block ℓ_{sc} for that thread.*

If the control mask and the predicate mask evaluate to true, then the thread executes the scheduled block actively. This is the essence of the activation mask $amsk^W$ defined in Definition 8.

Definition 8. *(Activation Mask) The activation mask (amsk) of a thread* $\mathtt{t} \in \mathcal{T}$ *in a P-LLVM execution state is the conjunction of the control mask and the predicate mask, i.e.* $amsk^W(\mathtt{t}) = pmsk^W(\mathtt{t}) \wedge cmsk^W(\mathtt{t})$.

Example: Thread Activation



```
A
iPtr = elemptr A tid
i = load iPtr
p = icmp uge i 0
notp = not p
br p B C

B
bPtr = elemptr B i
store bPtr x
br C

C
negi = sub -1 i
cPtr = elemptr C negi
store cPtr x
br D

D
ret x
```

(a) Block `C` has the predicate `notp`.

ℓ_{sc}	$cmsk^W$		$pmsk^W$		$amsk^W$	
A	1	1	1	1	1	1
B	0	1	1	1	0	1
C	1	1	1	0	1	0
D	1	1	1	1	1	1

(b) Schedule and mask function trace for a possible execution with two threads.

Figure 4.6.: A P-LLVM program with divergent control and a predicated basic block.

We show an example program in Figure 4.6 to give an understanding of how the control mask, predicate mask and activation factor into the execution. Figure 4.6a shows a small P-LLVM program that is executed by two threads. For the sake of the example, we will assume that the condition `p` evaluates to $[0, 1]$ for the threads, i.e. $\Delta^W(\mathtt{x})(\mathtt{t}_0) = 0$ and $\Delta^W(\mathtt{x})(\mathtt{t}_1) = 1$. Figure 4.6b shows the runtime schedule (column ℓ_{sc}) and the values of the control, predicate and activaton masks. Each row shows the control, predicate and activation mask according to the execution state after all preceding blocks of the runtime schedule have executed but before the scheduled block ℓ_{sc} of the current row executes. In the first row, we therefore see the predicate masks for the initial state. All blocks execute the entry block (`A`) and so the control mask is true for all threads. The block `A` is unpredicated, meaning its predicate value is true for all threads. The activation is defined as the conjunction of the control mask and the predicate mask and hence also all true.

The program contains a divergent branch in `A` with the first thread proceeding to block `C` while the second thread branches to `B`. Therefore, we see $cmsk^W = [0, 1]$ in the second row of the trace. The program also features a predicated basic block: the activation of block `C` depends on its predicate `notp`.

The block `C` sees its predicate `notp` evaluated differently for the threads. Therefore, when the block `C` is next on the schedule list (third row), we observe that $pmsk^W = [1, 0]$.

Example: Incremental If Conversion

Leveraging block predicates, a P-LLVM program can be if-converted incrementally while all intermediate stages have a defined semantics. Figure 4.7a shows the CFG of a P-LLVM program without predicates. If threads take different paths through the program, the control mask *cmsk* assures that blocks only actively execute if the block is on a thread's itinerary according to its control state.

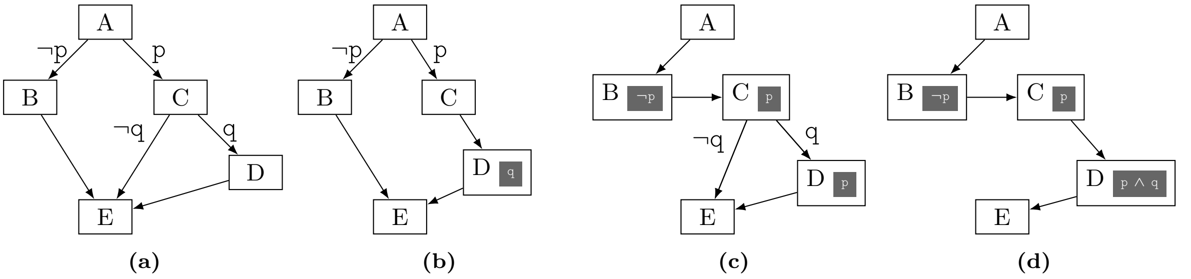


Figure 4.7.: Regardless of divergence in p or q, if the instructions in the blocks are the same, except for the conversion of ϕ nodes, all of these P-LLVM programs compute the same result.

In Figure 4.7b, the branch in `C` is if-converted and the former branch condition `q` predicates block `D`. The activation of block `D` is now the conjunction of the control mask and the value of `q`. More branches are if-converted in Figure 4.7c and Figure 4.7d further shifting control conditions into block predicates. In effect, when executed from the same initial state, the computed activations in all four CFGs are the same.

4.5. Semantic Interpretations

P-LLVM knows two semantic interpretations, `FR`(EEZING) and `OBL`(IVIOUS), that we touched on briefly in Section 3.6.1. If P-LLVM is interpreted under the `FR` semantics, instructions do not modify their result variable under passive execution. This interpretation models intuitive parallel execution where passive execution means that the values assigned to variables are completely unaffected by the passive execution. If an instruction executes with the `OBL` interpretation, the result variable of an instruction is assigned $\top$, the undefined value. This is unless P-LLVM language features such as the **total** modifier on instructions or shadow inputs on ϕ nodes are used. The `OBL` interpretation models what SIMD instructions really do: if a SIMD lane executes passively, it will still assign some value to that lane in the result register.

In the following definitions, the parameter $\mathcal{I}$ selects the semantic interpretation. There are two possible valuations: either $\mathcal{I} = \texttt{FR}$ for the freezing interpretation or $\mathcal{I} = \texttt{OBL}$ for the oblivious one.

4.6. Interacting with the Variable Mapping

The inference rules of the formalization use two formal functions to evaluate the value of syntactical P-LLVM values and to update the value bound to an identifier (*id*) in the variable mapping (Δ).

Evaluating Values. The formal function $\mathit{eval}^W\ \Delta^W\ \mathit{val}$ returns the value of the syntactical value *val*. The result is always a vector of proper values from $\mathcal{V}^W$. If *val* is an identifier, the returned values are looked up in the variable mappings Δ^W. Otherwise, *val* must be a constant and *eval* returns the value of this constant for all threads.

Updating the Variable Mapping. The formal function *updenv* assigns the elements of a vector of values to the same identifier in the W-vector of environments. We give the precise definition in Figure 4.8. Irrespective of the `FR` or `OBL` interpretation, an active instruction always updates its identifier. In `FR` semantics, the state of an identifier is retained if its instruction is not actively executing. In `OBL` semantics, an identifier is set to ⊤ in the same situation, meaning the former value of the identifier is replaced with an *undef* value. This is why the `OBL` interpretation of Figure 3.20a is not a faithful interpretation of the scalar code. When threads are forced to execute the loop body of the **`while`** loop despite already having left, they overwrite `i` and `x` to ⊤.

$$\boxed{\mathit{updenv_{\mathcal{I}}}^{W}\ \Delta^{W}\ \ \mathit{id}\ \mathcal{V}^{W}\ \ \mathit{amsk}^{W} \rightarrow \Delta^{W}}$$

$$\mathit{getNewValue}_{\mathcal{I}}\left(\mathit{amsk}^{W}, \Delta^{W},\ \mathit{id}, v^{W}\right)(\mathtt{t}) = \begin{cases} \top & \text{if } \mathcal{I} = \mathtt{OBL}\ \wedge\ \neg\mathit{amsk}^{W}(\mathtt{t}) \\ \Delta^{W}(\mathtt{t})(\ \mathit{id}) & \text{if } \mathcal{I} = \mathtt{FR}\ \wedge\ \neg\mathit{amsk}^{W}(\mathtt{t}) \\ v^{W}(\mathtt{t}) & \text{if } \mathit{amsk}^{W}(\mathtt{t}) \end{cases}$$

$$\frac{\forall \mathtt{t} \in \mathcal{T}.\left(\Delta'^{W}(\mathtt{t}) = \Delta^{W}(\mathtt{t})\left\{\ \mathit{id} \leftarrow \mathit{getNewValue}_{\mathcal{I}}\left(\mathit{amsk}^{W}, \Delta^{W},\ \mathit{id}, v^{W}\right)(\mathtt{t})\ \right\}\right)}{\mathit{updenv_{\mathcal{I}}}^{W}\ \Delta^{W}\ \ \mathit{id}\ v^{W}\ \ \mathit{amsk}^{W} \rightarrow \Delta'^{W}}\ [\text{UPD-ENV}]$$

Figure 4.8.: Update function for the identifier environment (Δ^{W}).

Vector Projection. For sake of legibility, we define the helper function *vec* as defined in Definition 9. It projects the value of variable across all threads into a vector. We omit the explicit Δ^{W} argument in *vec* when it is clear from the context, i.e. when referring to the vector projection of a variable at a specific program point.

Definition 9. *(Value projection) The vector projection $\mathit{vec}(\Delta^{W})(\mathit{id})$ projects the values of a variable across the threads to a vector value. $\mathit{vec}(\Delta^{W})(\mathit{id}) = \lambda\mathtt{t} \in \mathcal{T}.\Delta^{W}(\mathtt{t})(\mathit{id})$.*

4.7. Basic Block Semantics

The effect of executing a basic block for a thread array of W threads is captured by the basic block transition:

$$\ell_{sc} \vdash \left(\mathtt{M}, \Delta^{W}, {\ell_{in}}^{W}, {\ell_{next}}^{W}\right) \; \overline{\phi}\, \overline{c}\; tmn \rightarrow_{\mathcal{I}} \mathtt{M}' \; \Delta'^{W} \; i^{W}$$

A basic block ℓ_{sc} executes with an incoming execution state $\left(\mathtt{M}, \Delta^{W}, {\ell_{in}}^{W}, {\ell_{next}}^{W}\right)$. The subscript parameter $\mathcal{I}$ of the transition selects the semantic interpretation: FREEZING (FR) or OBLIVIOUS (OBL), discussed in Section 3.6. The execution produces a potentially altered memory state $\mathtt{M}'$, an environment with identifier assignments $\Delta^{W}{}'$ and a vector of successor indices i^{W}. The basic block transition is formalized by [BLS-W-STEP].

$$\frac{\begin{gathered} \ell_{sc} \;\; val_{msk} \;\; \overline{\phi} \;\; \overline{c} \;\; tmn \in \; prg \\ {\Delta_{pre}}^{W} = \textit{parallel-copy of } \; \overline{\phi} \;\; \text{using} \;\; {evalphi_{\mathcal{I}}}^{W} \;\; cmsk^{W} \;\; {\ell_{in}}^{W} \;\; \Delta^{W} \\ amsk^{W} \; = \; \left(cmsk^{W} \; {\ell_{in}}^{W} \; {\ell_{next}}^{W} \; \ell_{sc} \wedge eval^{W} \; {\Delta_{pre}}^{W} \;\; val_{msk}\right) \\ {computecmd_{\mathcal{I}}}^{W} \;\; amsk^{W} \;\; {\Delta_{pre}}^{W} \; \mathtt{M} \;\; c \rightarrow^{*} \Delta'^{W} \; \mathtt{M}' \\ i^{W} = evalterm^{W} \;\; amsk^{W} \;\; \Delta'^{W} \;\; tmn \end{gathered}}{\ell_{sc} \vdash \left(\mathtt{M}, \Delta^{W}, {\ell_{in}}^{W}, {\ell_{next}}^{W}\right) \rightarrow_{\mathcal{I}} \mathtt{M}' \; \Delta'^{W} \; i^{W}} \; \text{[BLS-W-STEP]}$$

The execution of a basic block by [BLS-W-STEP] proceeds in four stages.

Resolution of ϕ Nodes. First, the ϕ nodes are resolved. We use blue for these parts. Resolution of ϕ nodes follows standard parallel copy semantics [Appel, 1998]: That is, every ϕ node selects a new value from the incoming program state. Then, the identifiers of all ϕ nodes are assigned their new value. The function *evalphi* determines the result of individual ϕ nodes, which depends on the incoming block, the environment and the control mask. It is discussed in detail in Section 4.8.

Evaluating the Activation Mask. Second, The activation $amsk^{W}$ is computed as the conjunction of the control mask and the value of the block predicate. The instructions and the terminator depend on the activation of the basic block.

Executing the Instruction List. Third, the instructions are executed in sequence using the formal function *computecmd* and under the influence of the activation $amsk^{W}$. This stage is colored red. The semantics of instructions are further discussed in Section 4.9.

Evaluating the Terminator. Block execution finishes with the evaluation of the terminator using the function *evalterm*, which is defined in Figure 4.9. This yields the *successor index* i^{W}, which conceptually selects the control-flow edge to take from the currently executing block.

The successor index i for each thread is either an element of $\mathbb{N}$, the special value $\top$ or $\Box(r)$. If $i \in \mathbb{N}$, then the thread proceeds from ℓ_{next} in its current control state to the successor block ℓ_{dest} by the control-flow edge $(\ell_{next}, i, \ell_{dest}) \in \mathtt{E}$. If $i = \Box(r)$, the thread finishes program execution and yields the

$$\boxed{\mathit{evalterm}\ \mathit{amsk}\ \Delta\ \ \mathit{tmn} = i}$$

$$\mathit{evalterm}\ \mathit{amsk}\ \Delta\ \ \mathit{br}\ \ell = \begin{cases} 0 & \text{if } \mathit{amsk} \\ \top & \text{if } \neg\mathit{amsk} \end{cases}$$

$$\mathit{evalterm}\ \mathit{amsk}\ \Delta\ \ \mathit{ret}\ \ \mathit{val}_{\mathit{res}} = \begin{cases} \square(r) & \text{if } \mathit{amsk} \text{ given } r = \mathit{eval}\ \Delta\ \ \mathit{val}_{\mathit{res}} \\ \top & \text{if } \neg\mathit{amsk} \end{cases}$$

$$\mathit{evalterm}\ \mathit{amsk}\ \Delta\ \ \mathit{br}\ \ \mathit{val}_{\mathit{cnd}}\ \ell_{\mathit{0}}\ \ell_{\mathit{1}} = \begin{cases} 0 & \text{if } \mathit{amsk}\ \wedge\ \mathit{eval}\ \Delta\ \ \mathit{val} \\ 1 & \text{if } \mathit{amsk}\ \wedge\ \neg\mathit{eval}\ \Delta\ \ \mathit{val} \\ \top & \text{if } \neg\mathit{amsk} \end{cases}$$

$$\mathit{evalterm}^{W}\ \ \mathit{amsk}^{W}\ \Delta^{W}\ \ \mathit{tmn} = \mathit{map}^{W}\ \mathit{evalterm}\ \mathit{amsk}^{W}\ \Delta^{W}\ \ \mathit{tmn}$$

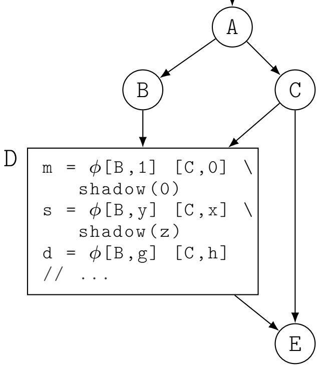
Figure 4.9.: Terminator semantics (*evalterm*).

return value r. Finally, if $i = \top$, then any reachable successor in the control-flow edge is an acceptable branch target.

4.8. Semantics of ϕ Nodes

P-LLVM augments ϕ nodes with an optional shadow operand. The purpose of this is to control the value of ϕ nodes under passive execution. The divergent loop transform presented in Section 9.1 leverages ϕ shadow operands to transform divergent loops into uniform loops. There, shadow ϕ nodes are used to selectively retain values on masked-off lanes.

The design of shadow semantics is motivated by the code transformations that happen in the vectorizer pipeline. To this end, Figure 4.10 jumps ahead, showing an example how ϕ nodes are lowered during if-conversion. The CFG on the left Figure 4.10a is completely if-converted on the right in Figure 4.10b.



```
m = φ[B,1] [C,0] \
   shadow(0)
s = φ[B,y] [C,x] \
   shadow(z)
d = φ[B,g] [C,h]
// ...
```

(a) CFG with ϕ nodes.

```
// control mask predicates
eBD = // ..
eCD = // ..

// select lowering of m
m = select(eBD, 1, 0)

// select lowering of s
s0= select(eBD, y, z)
s = select(eCD, x, s0)

// select lowering of d
d0= select(eBD, g, ⊤)
d = select(eCD, h, d0)
```

(b) After complete if-conversion.

Figure 4.10.: When if-converted, ϕ nodes lower to cascades of select instructions. The shadow input becomes the default value of the cascade.

When if-converted the control masks are computed explicitly. Each ϕ node is lowered into cascade of **select** instructions. Each **select** in the cascade selects an incoming value for the incoming control flow edge. If there is no **shadow** input, as is the case for `d`, the default value is undefined.

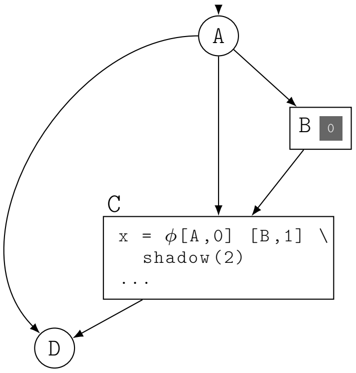
By adding a **shadow** input to a ϕ node that default value can be specified explicitly. We exploit this, for example, to convert control-flow into data without explicit path predicates. The variable `m` holds the explicit value of the control mask of block `D` in both the original CFG in Figure 4.10a and the if-converted version in Figure 4.10b. By virtue of the shadow input, we encode that the mask `m` is 0 if the control mask for `D` is false. We could make the CFG in Figure 4.10a part of a larger CFG that branches to `A` with divergent branches. Still, the invariant in Figure 4.10a that `m` is 0 if the control mask for `D` is false, even in that larger CFG, holds.

Formalization. The function *evalphi* returns the value of a ϕ node. It is used in the basic block transition to implement parallel-copy semantics for ϕ resolution (Section 4.7). In the absence of control divergence, ϕ nodes (shadow and non-shadow) behave as in regular, sequential LLVM IR.

$$\mathit{evalphi}_{\mathcal{I}}\ \mathit{cmsk}\ \ell_{in}\ \Delta\ \ \mathit{id} = \phi[\overline{\mathit{val}_j, \ell_j}] = \begin{cases} \mathit{eval}\ \Delta\ \ \mathit{val}_j & \text{if } \mathit{cmsk}\ \wedge\ \ell_j = \ell_{in} \\ \mathit{eval}\ \Delta\ \ \mathit{id} & \text{if } \neg\mathit{cmsk}\ \wedge\ \mathcal{I} = \texttt{FR} \\ \top & \text{if } \neg\mathit{cmsk}\ \wedge\ \mathcal{I} = \texttt{OBL} \end{cases}$$

The value assigned for non-shadow ϕ nodes under *passive* execution is $\top$. This is the same behavior as for variables assigned by *passive* instructions by *updenv* (see Section 4.6). In contrast, the semantics of shadow ϕ is invariant under the semantic interpretation.

$$\mathit{evalphi}_{\texttt{FR/OBL}}\ \mathit{cmsk}\ \ell_{in}\ \Delta\ \ \mathit{id} = \phi\overline{[\ \mathit{val}_j, \ell_j]}\ \mathbf{shadow}(\mathit{val}_s) = \begin{cases} \mathit{eval}\ \Delta\ \ \mathit{val}_j & \text{if } \mathit{cmsk}\ \wedge\ \ell_j = \ell_{in} \\ \mathit{eval}\ \Delta\ \ \mathit{val}_s & \text{if } \neg\mathit{cmsk} \end{cases}$$



```
x = ϕ[A,0]  [B,1] \
  shadow(2)
...
```

ℓ_{sc}	$\mathit{cmsk}^W(\ell_{sc})$		amsk^W		${\ell_{next}}'^W$		${\ell_{in}}'^W$	
A	1	1	1	1	D	B	A	A
B	0	1	0	0	D	C	A	B
C	0	1	0	1	D	D	..	..

$$\mathit{vec}(\texttt{x}) = [2, 1]$$

Figure 4.11.: Two threads approach a shadow ϕ node in block C. Columns marked $'$ refer to the resulting execution state.

We explain the mechanics of ϕ node resolution with the example in Figure 4.11. The CFG on the left is executed following the schedule on the right. The first column contains the scheduled block ℓ_{sc}, the next two columns show the control mask and activation for the scheduled block. Finally, the last two columns show parts of the execution state that result from executing the scheduled block, the resultant *next block* ℓ_{next} and the resulting incoming block ℓ_{in}. We forward to the point that C is scheduled. The second thread progresses from A to B and from there to block C. The ϕ node resolution is unaffected by the *activation* of its incoming block B. Hence, the second thread resolves the ϕ node to 1. The first thread goes from A right to D. The control mask for C is off when C is scheduled. The shadow ϕ node x resolves to its shadow input, yielding 2.

4.9. Semantics of Instructions

Each transition of *computecmd* consumes the first instruction off the instruction list and applies the effect of that instruction to the execution state. The transition rules are formally defined in Figure 4.12. After the last instruction of the list is executed, [W-CMD-EMPTY] returns the final environment and memory state, which is in turn matched in the inference rule [BLS-W-STEP]. The function *updenv* assigns the elements of the vector value v^W to the identifier *id* in the environments Δ^W of the threads.

$$\boxed{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \overline{c} \rightarrow \Delta^{W}\ \mathtt{M}}$$

$$\frac{}{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M} \rightarrow \Delta^{W}\ \mathtt{M}}\ \text{[W-CMD-EMPTY]}$$

$$\boxed{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \overline{c} \rightarrow \mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \overline{c}}$$

$$\frac{\begin{array}{c} c =\ \ \mathit{id} =\ \mathit{inst} \qquad \mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow \mathit{res}^{W}\ \mathtt{M}' \\ \mathit{updenv}_{\mathcal{I}}{}^{W}\ \Delta^{W}\ \ \mathit{id}\ \mathit{res}^{W}\ \mathit{amsk}^{W}\ \rightarrow \Delta'^{W} \end{array}}{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ c\,\overline{c} \rightarrow \mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta'^{W}\ \mathtt{M}\ \overline{c}}\ \text{[W-CMD-INST]}$$

$$\frac{\begin{array}{c} c =\ \ \mathit{id} =\ \mathit{inst}\ \mathbf{total} \\ \mathit{evalcmd}_{\mathcal{I}}{}^{W}\ 1^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow \Delta'^{W}\ \mathtt{M}' \qquad \mathit{updenv}_{\mathcal{I}}{}^{W}\ \Delta^{W}\ \ \mathit{id}\ \mathit{res}^{W}\ 1^{W}\ \rightarrow \Delta'^{W} \end{array}}{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \mathtt{M}\ c\,\overline{c} \rightarrow \mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta'^{W}\ \mathtt{M}'\ \overline{c}}\ \text{[W-CMD-TOTAL]}$$

$$\frac{c =\ \ \mathit{id} =\ \mathbf{mask}() \qquad \forall \mathtt{t} \in \mathcal{T}\left(\Delta'^{W}(\mathtt{t}) = \Delta^{W}(\mathtt{t})\left\{\ \mathit{id} \leftarrow a^{W}(\mathtt{t})\ \right\}\right)}{\mathit{computecmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \mathtt{M}\ c\,\overline{c} \rightarrow \mathit{computecmd}_{\mathcal{I}}{}^{W}\ \Delta'^{W}\ \mathtt{M}\ \overline{c}}\ \text{[W-CMD-MASK]}$$

Figure 4.12.: Rules for executing the instruction list of a basic block.

The **total** modifier has the effect of executing the command with all lanes enabled ([W-CMD-TOTAL]). This is similar to executing commands in an ISPC *unmasked* scope [Pharr and Mark, 2012]. We use total instructions to compute block predicates.

The actual instruction effect is modelled by *evalcmd*. The command **mask** is handled specially by [W-CMD-MASK] since it returns the activation amsk^W, so cannot use the **total** modifier, but assigns the result for all threads, like a **total** instruction.

We do not give an exhaustive definition of *evalcmd*. If an instruction is not explicitly specified in the following, each thread executes it independently according to regular LLVM semantics. Section 4.9.1 discusses the kind of guarantees given for explicit memory accesses. These guarantees transfer to unspecified instructions (memcpy), in so far applicable.

Syntactic Sugar. The P-LLVM language grammar as layed out in Figure 4.2 is designed for ease of formalization. We use a simplified syntax in actual code, as exemplified in Figure 4.13.

Note that the grammar of P-LLVM foresees all instructions to assign to a variable, i.e. to have the form `id = ...`. However, operations such as **store** or a call of function without return value do not yield a value. A program that contains uses of such an identifier is considered ill-formed. For the sake of legibility, we hence permit P-LLVM listings where the identifier is omitted in those cases, as shown in Figure 4.13a. We omit the return value in *ret* val_{res} in the same spirit when the CFG is that of a

```
store p v

// .. is synctactic sugar for ...

z = store p v
// (z must not be read)
```

(a)

```
a := add b c

// .. is syntactic sugar for ...

a = add b c total
```

(b)

Figure 4.13.: Simplified syntax used in P-LLVM code examples.

function without a return value. Figure 4.13b shows an example for the simplified syntax for **total**. We use the less verbose := notation for assignments to indicate that this is a **total** operation.

4.9.1. Memory

$$\boxed{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow \mathcal{V}^{W}\ \mathtt{M}}$$

$$\frac{\begin{array}{c}\mathit{inst} = \ \mathbf{alloca}\ \mathit{typ} \qquad s = \mathit{sizeof}(\mathit{typ}) \\ p^{W}, \mathtt{M}' = \mathit{newptr}^{W}\ \mathtt{M}\ s\ \mathit{amsk}^{W}\end{array}}{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow p^{W}\ \mathtt{M}'}\ \text{[W-INST-ALLOCA]}$$

$$\frac{\begin{array}{c}\mathit{inst} = \ \mathbf{load}\ \mathit{typ}\ \mathit{val}_{ptr} \\ p^{W} = \mathit{eval}\ \Delta^{W}\ \mathit{val}_{ptr} \qquad d^{W} = \mathit{read}^{W}\ \mathtt{M}\ \mathit{amsk}^{W}\ p^{W}\end{array}}{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow d^{W}\ \mathtt{M}}\ \text{[W-INST-LOAD]}$$

$$\frac{\begin{array}{c}\mathit{inst} = \ \mathbf{store}\ \mathit{val}_{ptr}\ \mathit{val}_{data} \qquad d^{W} = \mathit{eval}^{W}\ \Delta^{W}\ \mathit{val}_{data} \\ p^{W} = \mathit{eval}^{W}\ \Delta^{W}\ \mathit{val}_{ptr} \qquad \mathtt{M}' = \mathit{write}^{W}\ \mathtt{M}\ \mathit{amsk}^{W}\ p^{W}\ d^{W}\end{array}}{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \rightarrow \top^{W}\ \mathtt{M}'}\ \text{[W-INST-STORE]}$$

General function calls may modify the memory state in arbitrary ways. However, the **load**, **store** and **alloca** commands interface with the memory only through newptr^{W}, write^{W}, read^{W}. We informally require these functions to be well-behaved in the following sense.

Each invocation of $\mathit{newptr}^{W}\ \mathtt{M}\ s\ \mathit{amsk}$ returns a vector of pointers p^{W} where each element pointer refers to a mutable object in $\mathtt{M}'$. Each returned pointer is the base pointer to a memory buffer of size s. Allocation happens only for threads where the activation (amsk) is set, otherwise executing passively $p = \top$ and no memory is allocated for that thread. The allocated objects are *private*, that is each object is only accessible from one element of the pointer vector and pointers derived from them. The elements of a memory object are disjoint and non-aliasing.

Addressing elements of a memory object using **elemptr** yields pointers to element objects of the base pointer that alias with the base object. The **elemptr** command corresponds to `getelementptr` of LLVM IR.

$\mathit{read}^{W}\ \mathtt{M}\ \mathit{amsk}^{W}\ p^{W}$ performs a *gather* lookup of a vector of pointers (p^{W}) in the system state $\mathtt{M}$ for each lane whose activation is enabled (amsk^{W}). Each thread that reads from an uninitialized address obtains a $\top$ result.[3] $\mathit{write}^{W}\ \mathtt{M}\ \mathit{amsk}^{W}\ p^{W}\ d^{W}$ performs a *scatter* of a vector of values to a vector of pointers and returns the resulting system state $\mathtt{M}'$. Scattering the same value to the same address by multiple threads results in that value being stored at that address in the resulting state $\mathtt{M}'$. This happens in case of a typical *uniform* store. The behavior of scattering different values to overlapping memory ranges is not further specified.

[3] Otherwise loading from undefined memory addresses results in undefined behavior.

4.9.2. Function Calls

P-LLVM implements a flexible function call mechanism that encompasses horizontal operators, such as the *any* intrinsic, as ordinary functions.

$$\boxed{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \to \mathcal{V}^{W}\ \mathtt{M}}$$

$$\frac{\begin{array}{c} \mathit{inst} = \ \mathit{fnc}\left(\overline{\mathit{val}}\right) \\ \overline{\mathit{val}} = \ \mathit{val}_{\mathit{0}}, .., \ \mathit{val}_{n-\mathit{1}}, n \in \mathbb{N} \qquad \forall i \in \{\, 0, .., n-1\,\} \, . \left({p_i}^{W} = \mathit{eval}^{W}\ \Delta^{W}\ \ \mathit{val}_i\right) \\ f = \theta(\mathit{fnc}) \qquad \mathit{ret}^{W}, \mathtt{M}' = f\left({p_0}^{W}, .., {p_{n-1}}^{W}, \mathit{amsk}^{W}, \mathtt{M}\right) \end{array}}{\mathit{evalcmd}_{\mathcal{I}}{}^{W}\ \mathit{amsk}^{W}\ \Delta^{W}\ \mathtt{M}\ \mathit{inst} \to\ \mathit{ret}^{W}\ \mathtt{M}'}\ \text{[W-CMD-CALL]}$$

Figure 4.14.: General function call semantics.

Figure 4.14 defines the semantics of general function calls. The function definitions are available as mathematical function in the function map θ. A function call is performed by first evaluating the call parameters, looking up the function definition and then applying it to the execution state. The function explicitly takes the activation mask as a parameter. If the function entails a side effect on the memory state, this effect is thus properly masked. A function application results in a vector of return values ret^{W} and a potentially altered memory state M'.

Note that the underlying mathematical functions (f) operate on the vectors of parameter values across threads as a whole. By doing so, P-LLVM function can be horizontal operations whose result depends on the parameters of all threads.

P-LLVM requires mathematical definitions for the called functions and hence does not support calls to other P-LLVM functions and in particular recursion. Mathematical functions are sufficient for modelling the intraprocedural aspects of P-LLVM, which are the focus of this thesis. Calls to P-LLVM functions could be supplemented by adding a stackframe for function calls as VeLLVM does [Zhao et al., 2012].

Function calls with the total modifier amount to function call re-vectorization [Moreira et al., 2017].

Builtin Functions. We expect a reasonable runtime environment in θ, including common mathematical functions and the set of builtin functions shown in Figure 4.15. We omit the activation amsk^{W} and memory state M arguments for legibility understanding that memory is passed through and the activation ignored. If the return value is not a W-tuple, it is implicitly broadcast (for example, any actually returns Bit^{W} and each thread sees the same result value).

fnc	signature	$\theta(\mathit{fnc})$	informal description
`popcount`	$\mathit{Bit}^W \to \mathbb{N}$	$\lambda m^W.\left(\sum_{\mathtt{t}\in\mathcal{T}}.m^W(\mathtt{t})\right)$	Number of bits set in m^W.
`any`	$\mathit{Bit}^W \to \mathit{Bit}$	$\lambda m^W.\left(\exists_{\mathtt{t}\in\mathcal{T}}.m^W(\mathtt{t})\right)$	Whether any element of m^W is true.
`ballot`	$\mathit{Bit}^W \to (\mathit{Bit}^W)^W$	$\lambda m^W.\left[m^W, .., m^W\right]$	Broadcast the vector value m^W.
`thread_id`	$() \to \mathbb{N}^W$	$\lambda().\,[0, 1, .., W-1]$	The thread identifier.

Figure 4.15.: Builtin functions.

4.10. Well-formedness

P-LLVM is an extension of LLVM IR and inherits its well-formedness properties, e.g. that defining instructions must dominate their uses due to SSA form. P-LLVM poses additional well-formedness requirements that account for its data-parallel execution model and the addition of block predicates. The correctness of the divergence analysis, a fundamental analysis of the vectorizer, depends on the well-formedness of the P-LLVM program.

A central aspect to the correctness of P-LLVM programs is the extension of def-before-use to the predicated execution model. In standard SSA form, all variable definitions have to dominate their uses (or the incoming block in case of ϕ nodes). In P-LLVM, whether a variable is defined also depends on predicate under which the variable was assigned. We call this the definition mask of the variable, defined in Definition 10.

Definition 10. *(P-LLVM - Definition Mask) The definition mask of a constant is always* 1*. Otherwise, the value must be a variable and the following rules apply:*

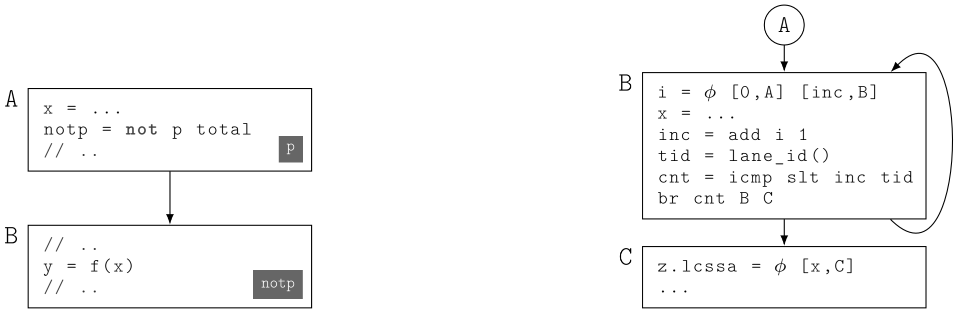
1. *If the variable is assigned by a non-total instruction, the definition mask is the activation mask of the block.*
2. *If the variable is assigned by a total instruction, then the definition mask is* 1*.*
3. *If the variable is assigned by a non-shadow ϕ node, then the definition mask is the conjunction of the control mask under which the ϕ evaluates and the definition mask of the selected incoming value.*
4. *If the variable is assigned by a shadow ϕ node, then if the control mask is true the same rules as for non-shadow ϕ nodes apply. Otherwise, the definition mask of the ϕ node is the definition mask of the shadow operand.*

Definition 11. *(P-LLVM Well-formedness)*

A P-LLVM program is well-formed if it has the following properties on top of the well-formendess requirements for regular LLVM IR.

1. ***(Dominating predicate)***
   If the predicate of a block is defined in another block, the block bearing the definition has to dominate the predicated block. If the predicate of a block is defined in the block itself, all preceding instructions in this block have to be total.
2. ***(Defined functions)***
   All function symbols fnc that occur in a P-LLVM program have to be defined in the function symbol map θ.
3. ***(Predicated def-before-use)***
   The requirement of SSA that a variable has to be defined before it is used (def-before-use) extends to predicated definitions and uses. In the execution of a instruction, the effect of the operation (system state `M` *or result value) may depend on the value of an operand that is given by an identifier val_{op}. Dependence of an operation on an operand is understood in the dynamic, semantical sense: does this instance of the operation show different behavior when the operand variable is modified? Whenever such a dynamic dependence to an operand exists, the definition mask of the user operation has to imply the definition mask under which the operand variable val_{op} was assigned.*

The well-formedness conditions for P-LLVM programs are layed out in Definition 11. The semantic nature of the predicated def-before-use may seem problematic since semantic properties cannot generally be verified at compile time. However, we presume well-formedness to exploit its properties in the vectorization system. The verification of well-formedness is a separate concern and out of the scope of this thesis.



```
x = ...
notp = not p total
// ..
```

```
// ..
y = f(x)
// ..
```

```
i = ϕ [0,A] [inc,B]
x = ...
inc = add i 1
tid = lane_id()
cnt = icmp slt inc tid
br cnt B C
```

```
z.lcssa = ϕ [x,C]
...
```

(a) The predicate of the using instruction (y) does not imply the predicate of the defining instruction (x).

(b) The variable x is set to the $\top$ under OBLIVIOUS and so the use by x.lcssa is ill-formed.

Figure 4.16.: Examples of ill-formed P-LLVM programs.

Figure 4.16 shows two cases of ill-formed P-LLVM programs, which demonstrate violations of the statute of predicated def-before-use (3. in Definition 11).

The first example in Figure 4.16a shows a violation in connection with block predicates. The variable x is defined under the block predicate p in block A. It is then used in block B under the negated predicate notp. Since x is not defined when it is used in block B, this is a violation under the assumption that the function f semantically depends on x.

The second example in Figure 4.16a shows a violation in connection with a divergent loop. The defining instruction of variable x is in the divergent loop that consists of the block B. The value of x is then read by the ϕ node z.lcssa outside the loop.

Consider that the program executes with two threads and that the first thread leaves the loop first and only in a later iteration the other thread. Under the FREEZING interpretation, the variable x is unchanged under passive execution for the first thread. When finally the block x is scheduled, both threads see the value of x as it was last defined in active execution, in their last active loop iteration.

Under the OBLIVIOUS interpretation however, the variable x is set to $\top$ for the first thread and under passive execution as B executes again after the first thread is already at C. We observe a violation of predicated def-before-use since the activation of the first thread was false when x was last assigned while the use in C again happens actively.

Generally, predicated def-before-use implies that programs with divergent loops are ill-formed under the OBLIVIOUS interpretation: Variables assigned in the divergent loop may be overwritten by passive execution when threads have left the loop. In that case, any (meaningful) use outside of the divergent loop makes the program ill-formed because active using instructions outside the loop then read a variable that was assigned in the divergent loop with a 0 definition mask.

4.11. Related Work

We place P-LLVM in the context of related work considering the following two angles. First, we compare P-LLVM with data-parallel languages and compiler IRs on the grounds of their expressiveness. Second, we look at literature outside of the domain of data-parallel programs. Section 5.4 discusses related work in synchronization and thread re-convergence.

4.11.1. Data-parallel Languages and Intermediate Representations

We compare the P-LLVM IR to other data-parallel languages and intermediate-representation with regards to their expressiveness. This includes horizontal operations and availability of predication.

Data-parallel Programming Languages. Data-parallel languages commonly do not offer ways to explicitly specify predicated execution [Pharr and Mark, 2012; Munshi, 2009; Leißa, 2017; Leißa et al., 2014; Reiche, 2018].

Horizontal operations (such as `any` and `ballot`) are usually modelled specially in data-parallel languages. This is true for HLSL `WaveActive*` kind of operations [Microsoft, 2018]. P-LLVM decouples the activation query (**mask**()) from the horizontal operation (`any`). As illustrated in Section 4.9.2, P-LLVM horizontal operations themselves are simply considered SIMD functions.

The CUDA programming language features a rich set of horizontal operations and features builtin functions that retrieve the current activation of a thread. Only the threads whose bit is set in the synchronization mask partake in the execution of the horizontal operation. With regards to horizontal operators the difference between P-LLVM and CUDA lies in thread synchronization, discussed further in Section 5.4.

Exchange Formats for Data-parallel Kernels. This category comprises standardized exchange formats for compute kernels.

SPIR-V [Kessenich et al., 2018] is the official intermediate representation for OpenCL and OpenGL shader programs [Kessenich et al., 2019], including the Vulkan API [Group, 2019]. Conceived to represent compute kernels, SPIR-V does not model predication either.

The NVIDIA CUDA [Nickolls et al., 2008] ecosystem provides two intermediate representations for GPU kernels. NVVM IR [NVIDIA, 2019b] is a dialect of LLVM IR for the representation of GPU compute kernels. NVIDIA PTX [NVIDIA, 2019a] is an instruction-set agnostic kernel IR for GPU compute kernels. PTX features predication and synchronization instructions to cater for the capabilities of various generations of NVIDIA microarchitectures. A total P-LLVM instruction corresponds to a PTX instruction without guard predicate in that both are computed for all lanes. PTX represents data-flow with virtual registers but without SSA form. However, both NVVM IR and PTX are primarily exchange formats, designed to simplify the design of frontends and to expose a stable interface to target NVIDIA GPUs.

LLVM IR. Several compilers for data-parallel compute languages are known to build on a data-parallel interpretation of standard LLVM IR. This includes the Intel OpenCL driver for CPUs [Rotem, 2011] as well as the more recent driver for Intel GPUs [Chandrasekhar et al., 2019]. Similarly, the GPU compiler of AMD is based on LLVM-IR [amd, 2019].

However, the stock transformations of LLVM were principally designed for sequential code. A special `convergent` attribute is used on function calls to indicate that the control dependences of the call must not be altered. However, deficiencies with this approach have been pointed out and it has been acknowledged [con, 2019] that a proper data-parallel semantics for a data-parallel reading of LLVM IR is required.

Data-parallel Compiler IRs. Farrell and Kieronska [1996] present a formal lock-step semantics for a structured language with divergent control statements. The approach models `FR` semantics, which inaccurately reflects the behavior of SIMD instructions in loops.

Karrenberg [2015] sketches a data-parallel, lock-step semantics for LLVM IR. The semantics retains variables under passive assignment as the `FR`EEZING mode in P-LLVM. Re-convergence is described to occur at the IPD.

The instructions of a P-LLVM program are specified in scalar terms, referring to the variable environment of each individual thread. There is a non-coincidental relation to work on Vector-Length Agnostic SIMD Instructions. This line of work encompasses *Vapor SIMD* [Nuzman et al., 2011] for compilers and *Liquid SIMD* [Clark et al., 2007] for SIMD ISAs. However, both approaches require explicit SIMD instructions and linearized control flow. There is no notion of control divergence as these languages are intended for direct lowering to machine code or, as is the case for Liquid SIMD, describe a SIMD ISA themselves.

Apart from data parallelism, several extensions for task-level parallelism have been proposed [Schardl et al., 2017; Khaldi et al., 2015]. Modelling parallelism at the level of multiple blocks or instructions at a time, task-parallel representations are too coarse-grained for use in a data-parallel vectorizer.

4.11.2. Predicated Execution and Passive Lanes

A considerable body of related work tackles the challenge of modelling predication in the compiler IR and in machine ISAs.

Hardware Support for Predication. Several contemporary SIMD ISAs support lane-wise conditional updates to the destination register. Values on masked-off lanes are passed through. Among their ranks are the AVX512 extension for X86 [Intel], the SX-Aurora TSUBSA Vector CPU [SXA, 2018], the RISC-V V extension [Alon Amid et al., 2019] and the AMD RDNA ISA [AMD, 2019]. SIMD instructions with conditional lane updates directly implement FREEZING semantics on the machine register level. Chains of conditionally-updating instructions to the same register form a dependence chain, a performance obstacle for out-of-order architectures [Chuang et al., 2003].

The ARM Scalable Vector Extension ISA [Arm, 2019] and AVX512 ISA [Intel] additionally implement destructive updates of the destination registers where masked-off lanes are set to zero. Other major SIMD ISAs, such as X86 AVX2, ARM NEON, Power AltiVec do not support predication at all. Lowering operations with FREEZING semantics for those ISAs therefore may require additional blend instructions to retain the masked-off lanes. In contrast, OBLIVIOUS semantics does not require any special hardware support as there are no requirements for masked-off lanes.

Predication in Compiler IRs. Several extensions to SSA Form have been developed to improve code generation for predicated architectures.

ψ-SSA [Stoutchinin and de Ferrière, 2001] uses ψ-nodes with explicit predication to represent select cascades. Importantly, they strictly require a *global* order over all input variables. Hence, code motion may break ψ-form making repairs necessary to re-constitute the normal form.

Gated-SSA augments ϕ nodes with explicit predicates [Tu and Padua, 1995a; Havlak, 1993; Ballance et al., 1990; Carter et al., 1999]. These predicates redundantly express the control-conditions under which an incoming block is taken at a ϕ node in regular SSA. The predicated Gated-SSA operators η,μ and γ are compatible with P-LLVM assuming they have total semantics. We could not find any extension to SSA form in related work nodes that captures the shadow operands of ϕ nodes provided in P-LLVM.

The Whole-Function Vectorizer by Karrenberg [2015] maintains block predicates in an external data structure, which is not considered part of the program.

LLVM IR. LLVM-based compilers for data-parallel codes take different approaches to work around the lack of native predication in LLVM IR. Intel [Rotem, 2011; Chandrasekhar et al., 2019] if-converts divergent branches early on and emulates predication with an operation-plus-select idiom. It is unclear how the Intel compilers maintain predicates for side-effect bearing instructions. On the other hand, the AMDGPU backend for LLVM [amd, 2019] maintains control flow until very late in the pipeline. Predication is implicitly expressed through control-flow idioms [Wahlster, 2018].

4.11.3. Formalization of Compiler IRs

VeLLVM [Zhao et al., 2012] provides a formal semantics for a sequential fragment of LLVM IR. Chakraborty and Vafeiadis [2017] presents a formal semantics for the specific issue of concurrent memory accesses only.

The use of the semantic modes, FREZZING and OBLIVIOUS, in P-LLVM is inspired by the modelling of imperative and functional semantic interpretations of *Linear IL* by Schneider et al. [2015]. While the works pursue different objectives, both systems use changing semantic interpretations of their intermediate language through the compiler pipeline.

4.11.4. Conclusion

To the best of our knowledge, P-LLVM is the first full-featured vectorizer IR for unstructured codes.

Existing data-parallel languages and compiler IRs ignore predication, except for the proprietary NVIDIA PTX stack. Public sources suggests that that the NVIDIA PTX compiler uses a data-parallel, predicated IR internally.

There exist various augmentations of SSA form for predication, none of which considers data-parallel execution and aspects of thread activation. The ψ-SSA and Gated SSA forms are otherwise orthogonal to P-LLVM.

Chapter 5.

Constraining the Runtime Schedule

P-LLVM programs execute with a runtime schedule, a sequence of block labels, which determines for every step of the program which block will execute next. This chapter presents greedy schedules, a restricted class of runtime schedules that mimic the control flow in the SIMD code that the system will eventually generate. Leaving the schedule space unconstrained would lead to imprecision in the divergence analysis, schedule-dependent program termination and inefficient SIMD code.

In the presentation of P-LLVM in Chapter 4, runtime schedules are without constraints: any sequence of block labels may be chosen as a runtime schedule for any execution of the program. However, runtime schedules that are theoretically possible in P-LLVM will never materialize in the SIMD code that RV generates. If the schedule space is left unconstrained, the divergence analysis would detect less uniform branches and instructions, causing ripple effects such as more branches being if-converted, spuriously divergent loops being transformed and more SIMD instructions being emitted. For example, the result of horizontal operators such as *any* is sensitive to the schedule but this sensitivity does not exist in the space of schedules that will actually materialize.

We call this restricted set of schedules greedy schedules. The class of greedy schedules is characterized by the three properties shown in Definition 12.

Definition 12. *(Greedy schedule) A runtime schedule $\overline{\ell_{sc}}$ is a* greedy schedule *for a given P-LLVM program if it has the following properties:*

- *The schedule makes* progress *(Section 5.1).*
- *All $\ell \in \mathbb{V}$ are* barriers *(Section 5.2).*
- *The schedule preserves termination (Section 5.3).*

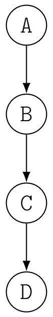
The remainder of this chapter motivates and formalizes the three properties we demand in greedy schedules, those are *Progress*, *Synchronization* and *Preservation of Termination*. The *RV* vectorization system assumes that all schedules are greedy schedules. Specifically, the divergence analysis (Chapter 6) and control-divergence analysis (Chapter 7) assume greedy schedules in their consideration of control-induced divergent effects.

5.1. Progress

Demanding progress achieves two things: First, it implies that when a block is scheduled at least one thread will have an active control masks for that block. This means that the runtime schedule cannot contain infinite sequences of blocks that do not advance the threads' control states. Second, a passive thread, i.e. $\ell_{next} = \top$, cannot make a branch that has a uniform branch condition divergent.

We discuss examples for both phenomena and finally describe and formalize the concept of progress for runtime schedules.

Example: Stalling. We left the schedule space entirely unconstrained: any block could be scheduled, even those that do not have any threads with an active control mask. This admits infinite schedules for P-LLVM programs that could execute to termination in a finite number of iterations. We show a practical example for this in Figure 5.1. Even for the loop-free CFG in Figure 5.1a, when not demanding an active control mask, there exists an infinite execution with an infinite schedule shown in Figure 5.1b. At the same time, there exists a terminating execution with a finite schedule as shown in Figure 5.1c.



(a) Example CFG.

ℓ_{sc}	$cmsk^W$		${\ell_{next}}'^W$	
A	1	1	B	B
D			B	B
B	1	1	C	C
B			C	C
B			C	C
...	...		...	

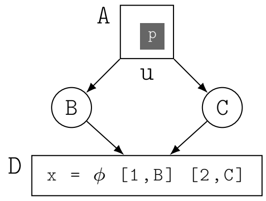
(b) Scheduling blocks without active control mask violates progress.

ℓ_{sc}	$cmsk^W$		${\ell_{next}}'^W$	
A	1	1	B	B
B	1	1	C	C
C	1	1	D	D
D	1	1	□	□

(c) Schedule with progress.

Figure 5.1.: Without progress, the schedule can pick blocks that no thread has an active control mask for (5.1b) - the execution stalls. Progress implies that at least one threads has an active control mask (5.1c).

Example: Spurious Control Divergence. Without further constraints, runtime schedules are prone to causing spurious control divergence in *uniform* terminators. Even if a branch condition evaluates to the same value for all active threads, threads executing the branch passively may still diverge, that is proceed to different successors of the branch. This is an issue because control divergence leads to inefficient SIMD code. An example for spurious divergence is shown in Figure 5.2.



	${\ell_{in}}'^W$		${\ell_{next}}'^W$		*Guides*	
ℓ_{sc}	$\mathtt{t}_0$	$\mathtt{t}_1$	$\mathtt{t}_0$	$\mathtt{t}_1$	$\mathtt{t}_0$	$\mathtt{t}_1$
A	A	A	C	⊤	{A}	{A}
B	A	B	C	D	{C}	{C}
C	C	B	D	D		
	...					

Figure 5.2.: Assume that $vec(\mathtt{p}) = [1, 0]$. The schedule on the right leads the second thread astray from the *uniform* terminator in A. The result is diverging control in a *uniform* branch condition and a divergent control-flow join in D. Last column with highlighted set: where spurious divergence is detected by Definition 14.

The branch condition in A is uniform. The thread $\mathtt{t}_0$ actively branches to block B. Since its activation is set to false by the block predicate, the thread $\mathtt{t}_1$ will have $\ell_{next} = \top$ after A has been scheduled: $\mathtt{t}_1$ will accept any of its successors as its next block. Hence, when C is scheduled next, $\mathtt{t}_1$ will proceed to block C. In conclusion, the threads reach a diverged control state despite the fact that the branch in A is uniform. When the schedule proceeds with C and finally D, the threads will reach D from two different incoming blocks. The ϕ node in D is hence not uniform since it resolves to different values for different threads.

Formalization. We resolve both issues, stalling (Figure 5.1) and spurious control divergence (Figure 5.2), by demanding every schedule to make *Progress*.

Stalling by itself can be resolved by requiring every scheduled block to have an active control mask for at least one thread. This follows as a corollary from the definition of progress, which we conclude in with Definition 14. The complexity in the definition of progress comes from the requirement to suppress spurious control divergence.

We first define the auxiliary concept of the *Guide Set* in Definition 13. We use Guide Sets to recognize when spuriously divergent control could occur as seen in Figure 5.2.

Definition 13. *(Guide Set) Let* $\sigma = \left(\mathtt{M}, \Delta^W, {\ell_{in}}^W, {\ell_{next}}^W\right)$ *be the current execution state. Let* $S \subseteq \mathtt{V}$ *be the set of all possible blocks to schedule from this state. Then, the* Guide Set *of a thread* $\mathtt{t} \in \mathcal{T}$ *for the state* σ *is defined as:*

$$Guides(\sigma, \mathtt{t}) = \left\{\, \ell_{next}(\mathtt{t}') \in S \;\middle|\; \ell_{in}(\mathtt{t}) = \ell_{in}(\mathtt{t}'), \mathtt{t}' \in \mathcal{T} \,\right\}$$

The *Guide Set* of a thread is the set of schedule-able blocks that other threads executing the same terminator branch to *actively.* Spurious control divergence occurs when there are two threads at a uniform branch: One of them leaves the branch in a control state where it will accept any successor ($\ell_{next} = \top$) and the other has a specific successor block (ℓ_{next} is a block label). In Figure 5.2 B is not in the guide set of $\mathtt{t}_1$ when B is scheduled. We use the concept of the *Guide Set* to define the notion of *Progress* of a scheduled block in Definition 14.

Definition 14. *(Progress) Let* σ *be the current execution state. Then, scheduling* $\ell_{sc} \in \mathtt{V}$ *makes* Progress *if one of the following holds:*

1. *There exists a* $\mathtt{t} \in \mathcal{T}$ *such that* $\ell_{next}(\mathtt{t}) = \ell_{sc}$.
2. *For all threads* $\mathtt{t} \in \mathcal{T}$ *such that there exists an edge* $\ell_{in}(\mathtt{t}) \to \ell_{sc}$*, it holds that* $Guides(\sigma, \mathtt{t}) = \{\ \}$.

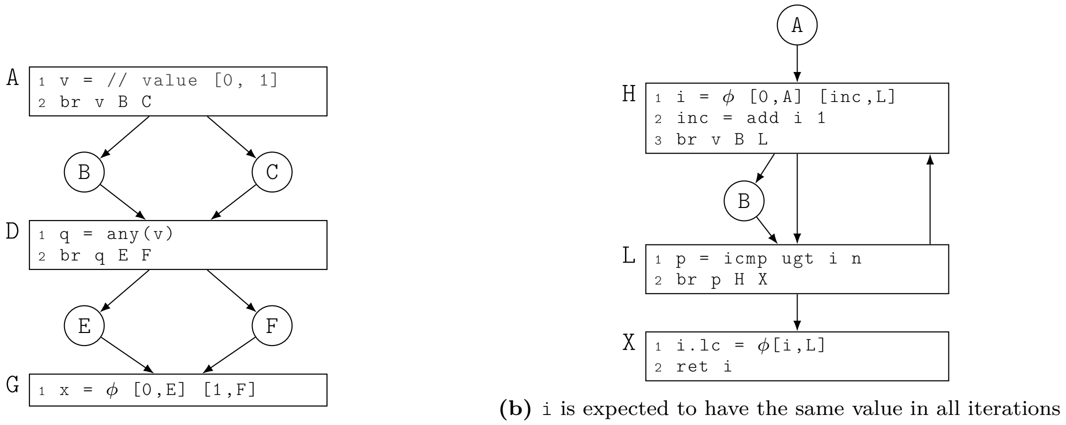
A schedule makes progress for a P-LLVM program and (initial) execution state, if all block executions with the schedule make progress.

All schedules we consider, with the exception of the counter example in Figure 5.2, make progress. In Figure 5.2, $Guides(\sigma, \mathtt{t}_1) = \{\,\mathtt{C}\,\}$ after the first line of the trace. Scheduling B next thus does not progress since for the first thread $\ell_{next}(\mathtt{t}_0) \neq \mathtt{B}$. Hence, neither condition of Definition 14 is satisfied.

5.2. Synchronization and Barriers

Even if only considering progressing schedules, the choice of schedule for a given program may have profound impact on behavior of the program. The outcomes of function calls and shared memory accesses may depend on the activation. The activation depends on the control mask, which in turn depends on the scheduled block. Runtime schedules may also differ in how often they execute a given block. These factors can cause runtime schedules to manifest in the P-LLVM execution state. The semantics of a P-LLVM program is hence schedule dependent.

Consider the P-LLVM programs in Figure 5.3. Executing the program in Figure 5.3a with the schedule $\overline{\ell_{sc}} = \mathtt{ABCDEFG}$ yields $vec(x) = [0, 0]$. However, using the schedule $\overline{\ell_{sc}} = \mathtt{ABDCDEFG}$, we see $vec(x) = [1, 0]$. This second schedule stalls the second thread waiting for C to be scheduled while the first threads executes the any call in D alone. Since v is 0 for the first thread, any(v) is also 0 and the first thread branches to F next. When D is scheduled again after C, only the second thread executes the any call actively as the first thread is already past D. Since for the second thread v and thus any(v)



```
A:
v = // value [0, 1]
br v B C

D:
q = any(v)
br q E F

G:
x = φ [0,E] [1,F]
```

```
H:
i = φ [0,A] [inc,L]
inc = add i 1
br v B L

L:
p = icmp ugt i n
br p H X

X:
i.lc = φ[i,L]
ret i
```

(a) x is expected to have the same value for all threads.

(b) i is expected to have the same value in all iterations of H.

Figure 5.3.: 5.3a: Horizontal operations can manifest the schedule in the execution state. 5.3b: The schedule determines whether values are *uniform*, e.g. the same for all threads.

evaluate to 1, the second thread branches to E. When lastly G is scheduled, both threads are active and the first thread approaches from F and the second thread from E, the ϕ node is evaluates and results in $vec(x) = [1, 0]$.

Besides semantics, the choice of schedule has implications on the SIMD code one may generate for a P-LLVM program. If a variable always holds the same value across threads, it is *uniform*, the SIMD code may represent it as a scalar value. If the value may differ between threads, it is *varying* and the SIMD code has to represent it as a proper SIMD variable.

In Figure 5.3b the value of i is *uniform* if the threads execute the loop iterations in lock step. That is there are never two threads in different loop iterations. Yet, without any constraints on the schedule space, two threads may be in different loop iterations at the same time, actively executing the assignment to i, which as a result has to be considered *varying*. For example, assume that $vec(v) = [1, 0]$ and the schedule $\overline{\ell_{sc}} =$ AHLHBLHLX When H is scheduled for the first time, the first thread branches to B. The second thread actively executes L and H again with the first thread passive. When B is scheduled, both threads actively execute L and H again. Consider this third scheduling of H. At this point both threads are active. The first thread has only once actively executed H before and sees i with value 0. However, it is the third execution of H for the second thread and i was incremented twice since. Therefore, the second thread sees i set to 2. In short, both threads are active and see a different value for i. Hence, when such a schedule is legal, i is *varying* and has to be widened into a SIMD instruction. If we disallow schedules that lead threads to be in different loop iterations at the same time i can be considered *uniform* and can be kept scalar.

As discussed before, the unconstrained set of live schedules allows certain schedule-dependent behaviors. Restricting the schedule space has the advantage that the behavior of the P-LLVM program becomes deterministic in more cases as more schedule-dependent behaviors are ruled out. We restrict the schedule space by enforcing schedule constraints. The schedule constraints take the form that certain blocks are required to be barriers, they may be only be scheduled if there are no other blocks reaching them without back edges that could be scheduled instead. We define the notion of reachability underlying these schedule constraints in Definition 15. Definition 16 further defines the set of blocks that have a forward reaching path to a given block.

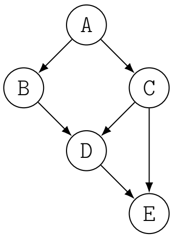
Definition 15. *(Forward reaching path) A path $\pi \in a \rightarrow^* z$ is called* forward reaching, *if $\forall e \in \pi . e \notin$ Backedges. Informally speaking, the path π does not take a back-edge.*

Definition 16. *(Forward reachability) We define the set of forward reaching blocks of a block* $z \in \mathtt{V}$ *as* $\mathit{fwdreaching}(z) = \left\{ a \in \mathtt{V} \;\middle|\; \exists \pi \in a \to^{+} z.\pi \text{ is forward reaching} \right\}$.

Definition 17 characterizes the notion of synchronization, the kind of behavior we enforce to restrict the schedule space. Intuitively, block execution is synchronizing if no passive thread reaches this block without taking a backedge.

Definition 17. *(Synchronization) Let* $\left(\mathtt{M}, \Delta^{W}, {\ell_{in}}^{W}, {\ell_{next}}^{W}\right)$ *be the current execution state. Then, scheduling* $\ell_{sync} \in \mathtt{V}$ *is* synchronizing, *if* $\forall \mathtt{t} \in \mathcal{T}.\ell \in \mathit{fwdreaching}(\ell_{sync}) \implies \neg\left(\mathit{cmsk}\ \ell_{in}(\mathtt{t})\ \ell_{next}(\mathtt{t})\ \ell\right)$.

Synchronization is a dynamic property of the execution. We give an example for synchronization in Figure 5.4.



(a) A CFG.

ℓ_{sc}	$amsk^{W}$		${\ell_{next}}'^{W}$		$\mathit{fwdreaching}(\ell_{sc})$	
A	1	1	B	C	{ }	*sync*
B	1		D		{ A }	*sync*
D	1		E		{ A, B, C }	
C		1		D	{ A }	*sync*
D		1		E	{ A, B, C }	*sync*
E	1	1	□	□	{ A, B, C, D }	*sync*

(b) Execution trace. The last column indicates whether the scheduled block is *synchronizing.*

Figure 5.4.: D is not a barrier. When D is scheduled for the first time, the *next* block of the second thread reaches D.

When the block D is scheduled for the first time, the first thread executes it *actively.* At the same time, the second thread is still pending with its next block at C. As C ∈ *fwdreaching*(D) this first execution of D is hence not synchronizing. All other schedule transitions in the example are synchronizing.

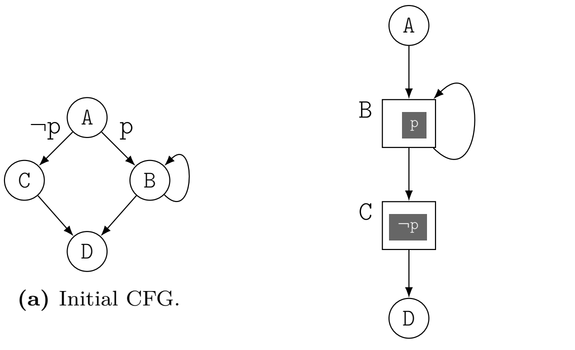
We demand for greedy schedule that all scheduled blocks are always synchronizing. We formalize this by saying that all blocks are *barriers* as of Definition 18.

Definition 18. *(Barrier) A block* $\ell_{barrier} \in \mathtt{V}$ *is a barrier in the execution of a P-LLVM program, if whenever* $\ell_{sc} = \ell_{barrier}$ *in the transition* $\ell_{sc}\overline{\ell'_{sc}}\ {\ell_{in}}^{W}\ {\ell_{next}}^{W}\ \Delta^{W} \xrightarrow{\ell_{sc}}\!\!\!\twoheadrightarrow_{\mathcal{I}} \overline{\ell'_{sc}}\ {\ell_{in}}^{W'}\ {\ell_{next}}^{W'}\ \Delta^{W'}$ *then* $\ell_{barrier}$ *is synchronizing in the left-hand side state* ${\ell_{in}}^{W}\ {\ell_{next}}^{W}\ \Delta^{W}$ *of the transition.*

5.3. Termination Preservation

Predication can cause threads to get stuck in a loop that no active thread is executing. There can then be an infinite schedule that keeps spinning that cycle and another schedule that would lead threads out of the loop. If there exists a (finite) runtime schedule that leads all threads to a terminated state, we disallow all (infinite) runtime schedules.

Consider Figure 5.5. If-conversion may transform the CFG shown in Figure 5.5a to the CFG in Figure 5.5b, if p is the divergent branch condition in A. Assume that p is 0 in the entry block for both threads. There now exist two runtime schedules, both of them make progress and are synchronizing in every scheduled block: The first schedule, ABCD, leads both threads to a terminated state. The second schedule starts in A and then schedules B infinitely often as shown in Figure 5.5c. Both threads start in



(a) Initial CFG.

(b) 5.5a after if-converting the branch in A.

ℓ_{sc}	$cmsk^W$		$amsk^W$	
A	1	1	1	1
B	1	1		
B	1	1		
B	1	1		
...	...	...	...	

(c) Execution of 5.5b assuming $vec(\mathrm{p}) = [0, 0]$.

Figure 5.5.: Demanding progress is not sufficient to preserve program termination. 5.5b is a realistic CFG created by if-converting 5.5a. The execution shown in 5.5c has an infinite schedule and the program never terminates. Yet, the schedule consisting in the sequence: A, B, C and finally D leads to termination.

A and branch to B. The predicate in B is p and therefore the threads always execute B passively. This means that the successor index for the branch in B is always $i^W = \top$, i.e. both threads accept accept any successor of B as next block. It is thus consistent with progress to schedule B infinitely often.

We require that all non-terminating schedules are illegal, if a legal terminating schedule exists. By that token the schedule in Figure 5.5c is illegal.

5.4. Related Work

We discuss the aspect of data-parallel scheduling and synchronization in related work separately with regards to hardware architectures and programming languages for SIMD programming.

Data-parallel Hardware. Historically, Levinthal and Porter [1984] present the CHAP SIMD ISA with structured control flow elements (ifthen, while, ...). They introduced the notion of the divergence stack. Each entry of the stack holds a region (single-entry, single-exit part of the program) and a thread mask, designating all threads that need to execute that region. Upon executing a divergent control-flow instruction the successor regions are put on the stack with the corresponding activation masks for the threads. The hardware executes both cases and control-flow re-converges after the control-flow instructions.

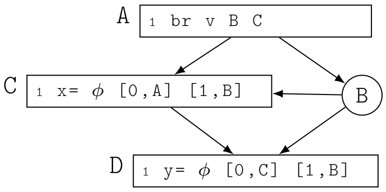


Figure 5.6.: Since D is the IPD of A, when control diverges in A the block C may be scheduled twice when a divergence stack is used.

Fung et al. [2007] generalized the divergence stack for unstructured control flow. The re-convergence point is the immediate post dominator (IPD). For example, in Figure 5.6, if the branch in A diverges

C and B are put on the divergence stack with complementary thread masks. Since control-flow only reconverges at block D, the IPD of A, the block C may be scheduled twice, once for threads branching directly from A to C and once again for threads originating from B. Note that this means that the IPD model of reconvergence is incompatible with OBLIVIOUS semantics: when a block is executed for the second time the threads that already passed it in the first execution see their variables set to $\top$. In contrast, greedy schedules guarantee that C only executes once and for all threads that reach it.

The IPD model of reconvergence became a mainstream staple with its implementation in NVIDIA GPUs. Starting with the NVIDIA Tesla microarchitecture in 2008, thread re-convergence in CUDA was stack-based [Lindholm et al., 2008; Habermaier and Knapp, 2012]. From there, the assumption of IPD reconvergence found its way into related work on the analysis data-parallel of kernels [Coutinho et al., 2011; Sampaio et al., 2013; Alur et al., 2017].

The NVIDIA Volta microarchitecture breaks with the IPD model and introduces an independent program counter for every thread [NVIDIA, 2017, Fig. 21]. Thread re-convergence is not guaranteed to happen at any time unless explicit synchronization instructions enforce it [NVIDIA, 2017, Fig. 22]. The synchronization model of CUDA changed starting with CUDA 9.0. Before CUDA 9.0, the horizontal operations implicitly accounted for the current activation mask as in HLSL. Since CUDA 9.0, every horizontal operation is an explicit synchronization point and horizontal operations take an additional synchronization mask argument. This execution models persists with the NVIDIA Turing microarchitecture [NVIDIA, 2018]. There is no public documentation on how and to what extent the CUDA driver inserts re-convergence points. The optimizing code generator in the CUDA compiler, the part that translates PTX to the native GPU ISA, is proprietary. The PTX manual only states that "For divergent control flow, the optimizing code generator automatically determines points of re-convergence" [NVIDIA, 2019a, Section 9.5].

Likewise, recent Intel HD and Iris Graphics GPUs break with the IPD reconvergence model [Chandrasekhar et al., 2019]. Threads may reconverge before the IPD but a full join of all threads will occur at the IPD.

In contrast, the RDNA ISA for AMD GPUs is a SIMD ISAs with explicit predication [AMD, 2019]. The earlier AMD GCN ISA also featured a software-controlled re-convergence stack, which implements IPD re-convergence [AMD, 2017].

The greedy schedule for P-LLVM programs enforces re-convergence at the statically earliest point, which might lie before the IPD. The independent thread scheduling of NVIDIA Volta architectures is not fully representable as P-LLVM schedule spaces. If the *cmsk* of a thread accepts the next scheduled block, it will execute it in any case. It is not possible for a subset of the threads that accept a scheduled block to decide against executing it. However, for lack of public documentation, it is not clear whether that is a scheduling decision that NVIDIA microarchitectures would actually make.

P-LLVM offers a form of "Implicit Warp-Synchronous Programming" (in jargon of CUDA), which means allowing data-parallel program whose correctness depends on the runtime schedule. In P-LLVM, warp synchronization is made explicit by the set of constraints put on the schedule space.

Collange [2011b]; Diamos et al. [2011] propose data-parallel microarchitectures, which implement greedy schedules in hardware. The block priorities are statically determined by the ordering of the blocks in machine code. In comparison, P-LLVM does not assign fixed priorities and allows for more greedy schedules for a given program than those architectures. In P-LLVM runtime schedules, static block priorities may be expressed by constraining the schedule space. Constraints may, for example, enforce schedules that prefer B over C when both blocks have pending threads).

ElTantawy et al. [2014] present multi-path scheduling, a thread scheduling mechanism for data-parallel programs. There is a one-to-one correspondence between the schedules in multi-path scheduling and P-LLVM runtime schedules.

Data-parallel Languages. Data-parallel languages that target a broad set of targets (NVIDIA GPUs, AMD GPUs, CPU SIMD ISAs), still follow the IPD re-convergence model. The includes the HSA IL [Glossner et al., 2015] and SPIR-V [Kessenich et al., 2018] intermediate languages.

OpenCL [Munshi, 2009] is thread centric data-parallel programming language. The OpenCL specification mentions divergent control flow but does not mention re-convergence points at all. There is support for explicit synchronization across threads through the `work_group_barrier` statement (former `barrier`). However, barriers in OpenCL have to be placed in uniform control flow.

The OpenCL Shader Languages (GLSL) is a structured, data-parallel compute language for use in computer graphics [Kessenich et al., 2019]. The languages features no goto statement but allows switches and break statements. The specification loosely defines a notion of uniform control-flow, "Uniform control flow (or converged control flow) occurs when all invocations in the invocation group execute the same control-flow path". Crucially, the specification further states that "If control flow is uniform upon entry into a selection or loop, and all invocations in the invocation group subsequently leave that selection or loop, then control flow reconverges to be uniform.". We interpret this as meaning that re-convergence will occur at the earliest possible point, implying greedy scheduling.

SPIR-V [Kessenich et al., 2018] allows explicit specification of synchronization points at post dominators of divergent loops (`OpLoopMerge`) or branches (`OpSelectionMerge`). Threads are only guaranteed to re-converge at the *unique* associated merge block of the aforementioned operations. When translating GLSL shaders to SPIR-V, divergent branches have to use explicit synchronization. However, SPIR-V synchronization primitives are static, whereas the GLSL specification explicitly mentions reconvergence of "invocations" at runtime. The SPIR-V lowering of GLSL is thus not compatible with the GLSL specification, regarding synchronization.

The SPIR-V specification uses the term *inactive invocation* to refer to a concept inverse to the activation in P-LLVM. However, the SPIR-V specification does not define that term further.

Several data-parallel languages with lock-step semantics have been proposed for vectorization [Pharr and Mark, 2012; Leißa, 2017; Leißa et al., 2014; Reiche, 2018]. These languages do not allow unstructured divergent control flow (goto statement). Re-convergence is defined to occur after a diverging control-flow statement.

[Ngo, 1995] introduces the `DOVEC` statement to structured Fortran code, which models lock-step execution of an outer loop. As for other structured languages, re-convergence occurs at the exit of a structured control flow statement. OpenMP [Klemm et al., 2012] defines a lock-step execution mode for loops and functions with the compiler directives `#pragma omp simd` and `#pragma omp declare simd`. However, the OpenMP standard neither defines lock step nor re-convergence with regards to this SIMD mode.

Conclusion. Only recent work in the domain of computer architecture breaks with the paradigm that control re-converges at the immediate post dominator. Surveying related work, we found that this change in GPU ISAs has not yet made it into the research literature on data-parallel IRs. P-LLVM runtime schedules fall in-between IPD reconvergence and the ine-grained synchronization offered by recent NVIDIA microarchitectures.

Chapter 6.

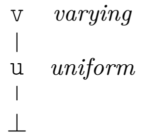
Divergence Analysis of P-LLVM Programs

The *Divergence Analysis* (DA) computes which variables and stack objects in a program are uniform, that is they always hold the same value across the active members of the thread array. Divergence analysis plays a central role in the vectorizer: Divergence analysis can prove a P-LLVM program control uniform and if this is not the case, which branches are divergent. This information guides the divergent loop transform and partial control-flow linearization to bring the program into control uniform form. Finally, the widening stage leaves uniform instructions and stack objects scalar, a crucial optimization for efficient SIMD code.

The divergence analysis presented in this chapter is the first to consider all divergence effects in P-LLVM programs. Different to prior work, we give a thorough treatment of stack-allocated objects, consider aspects of thread activation and account for the data-parallel **total** and **shadow** modifiers of P-LLVM.

6.1. Vector Shapes and the Basic Divergence Lattice

The Divergence Analysis is a data flow analysis.The analysis assigns to each program variable a so-called vector shape, which is an element of the abstract divergence lattice. In its basic form, the divergence lattice knows three different abstract states, shown in Figure 6.1: Uniform (u) and Varying (v) and a bottom element (⊥). Other more sophisticated Divergence Lattices, such as the one introduced in Chapter 10, are refinements of this basic lattice.



$$\gamma(\mathtt{v}) = \left\{ \lambda \mathtt{t} \in \mathcal{T}.y(\mathtt{t}) \;\middle|\; y \in \mathcal{V}^W \right\}$$

$$\gamma(\mathtt{u}) = \{ \lambda \mathtt{t} \in \mathcal{T}.y \mid y \in \mathcal{V} \} \qquad \gamma(\bot) = \{ \}$$

Figure 6.1.: Vector shapes and partial order of the basic divergence lattice (left) and its concretization function γ (right).

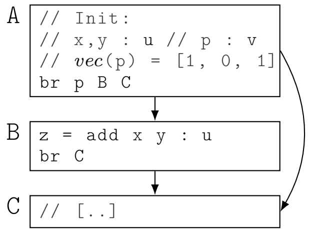
We refer to the elements of a divergence lattice as vector shapes. We denote by x : *s* that the variable x has the vector shape *s*.

If an instruction has a *uniform* vector shape, all threads that actively execute it see the same value in the result variable. Otherwise, they are called *varying.* The concretization function γ maps vector shapes to sets of observable vectors. That is, if x has the vector shape $\mathtt{x}^\sharp$ then for each reachable execution state if it holds that $vec(\mathtt{x}) \in \gamma(\mathtt{x}^\sharp)$[1].

```
// Init:
//   y,z : v
//   A,b,c : u

x = add y z : v
a = xor b c : u
w = mul x a : v
ptr = elemptr A w : v
z = load ptr : v
```

(a) Vector shapes in straight line code.



```
// Init:
// x,y : u // p : v
// vec(p) = [1, 0, 1]
br p B C
```

```
z = add x y : u
br C
```

```
// [..]
```

(b) Variable z has the same value for all threads where p is true.

Figure 6.2.: Representation of vector shapes in P-LLVM code.

The concretization function γ is appropriate in uniform control flow and in the absence of predication. Consider the straight line code in Figure 6.2a. If a variable in the block, e.g. b, is uniform its value is same value for all threads without exception.

6.2. Masked Concretization

As we defined it, the concretization function γ is not generally applicable in the context of divergent control flow. We show an example with a divergent branch in Figure 6.2b. Assume that the branch condition p evaluates to $vec(\mathtt{p}) = [1, 0, 1]$ for $W = 3$. When the program has finished, z has the same value for the first and third thread. The second thread never actively executes B and neither initializes the variable z. The variable z is conceptually uniform because it does have the same value for all threads

[1] $vec(\mathtt{x})$ is the vector of values assigned to x across all threads in a program state.

that actively execute the add that defines it. However, this is not reflected by the concretization function γ. Therefore, we introduce the concept of the *masked concretization*.

$$weaken(g \in \mathcal{T} \to \mathcal{V}, m \in Bit^W) = \lambda \mathtt{t} \in \mathcal{T}. \begin{cases} \top & \text{if } \neg m(\mathtt{t}) \\ g(\mathtt{t}) & \text{if } m(\mathtt{t}) \end{cases}$$

$$\gamma'(x^\sharp) = \lambda m \in Bit^W. \left\{ \, weaken(g, m) \;\middle|\; g \in \gamma(x^\sharp) \, \right\}$$

Figure 6.3.: The masked concretization (γ') is weakened to allow any value on inactive lanes. Different to the regular concretization (γ), it is valid in the presence of divergent control flow and predication.

The masked concretization (γ') defined in Figure 6.3 limits the expressive power of vector shapes to those lanes assigned with an enabled definition mask. It does so by mapping vector shapes to functions that map concrete definition masks, bit vectors, to sets of concrete vectors. This has the consequence that vector shapes can only be interpreted in the context of a specific definition mask.

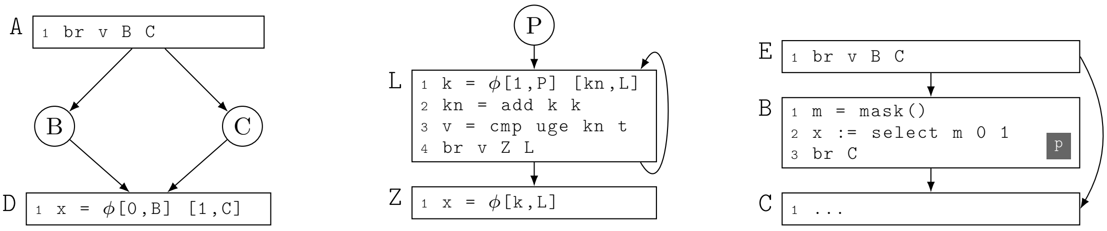
Coming back to the example in Figure 6.2b, we concretize the uniform shape of x through $\gamma'(z^\sharp)$. The result is a function from activation vectors to sets of concrete vectors. Knowing that *vec*(p)= $[1, 0, 1]$, we obtain the set of concrete values of x through $\gamma'(z^\sharp)([1, 0, 1])$. In each vector in the concrete set, the first and third lane are tied to the same value whereas the second value is the undefined value.

As discussed in Section 4.10, any operation with a semantic dependence on z has to execute under a definition mask that implies the definition mask under which z was defined. Therefore, if Figure 6.2b is a well-formed P-LLVM program, then any semantically relevant use of the variable z in an operand position will see the same value for z.

6.3. Control Divergence

The result shape of an instruction depends on the vector shapes of its operands. If a branch condition is varying, threads executing the branch may proceed to different successor blocks. The branch is then called *divergent* and is said to cause *control divergence*.

Control divergence induces non-uniformity in variables and instructions. The vector shape of instructions and variables thus depends on the divergence of terminators. We categorize this dependence in three categories:



```
br v B C
```

```
x = φ[0,B] [1,C]
```

(a) Divergent control join.

```
k = φ[1,P] [kn,L]
kn = add k k
v = cmp uge kn t
br v Z L
```

```
x = φ[k,L]
```

(b) Temporal divergence.

```
br v B C
```

```
m = mask()
x := select m 0 1
br C
```

```
...
```

(c) Varying activation.

Figure 6.4.: Control divergence influences vector shapes. In all cases assumed divergence in `v` makes `x` varying.

Sync Dependence. After threads diverge at a terminator they may reach a ϕ node from different incoming blocks at the same time (Figure 6.4a). In this case, the ϕ node may produce a varying value, even though its value operands are uniform. This dependence relation between blocks of divergent control and blocks of joining control is referred to as *Sync Dependence* [Coutinho et al., 2011]. We write $z \in \mathit{sdep}(a)$, if control divergence in a may cause z to be reached by divergent threads.

Temporal Dependence. An instruction that is uniform inside its defining loop may appear varying to users outside the loop (Figure 6.4b). Loop live-out variables may be re-defined in every loop iteration. The observable vector shape in each iteration may be uniform. If threads leave the loop in different iterations the observed live-out vector may be indeed varying. In Figure 6.4b, `k` is uniform inside the loop and holds a different value in each iteration. The live-out vector of values of `k` reaching the LCSSA ϕ in `Z` may be a blend of `k` from different iterations.

We denote by $e \in \mathit{tdep}(a)$ that control divergence in a may cause temporal divergence in the loop exit e. We require Loop-Closed SSA form [Pop, 2006] for simplicity so that temporal divergence can only affect ϕ nodes in immediate loop exits.

Activation Dependence. The vector shape of an instruction may depend on its activation. Consider the variable `m` in Figure 6.4c. Since `mask()` simply returns the activation bit, the shape of `m` depends on `v` and on `p`. If `v` makes the branch in `E` divergent, the activation of `B` becomes varying. Further, irrespective of whether `v` is uniform, if the block predicate `p` is varying, `m` is varying as well. A static approximation of activation dependence is computed directly in the divergence analysis algorithm.

Chapter 7 presents the algorithm that we use to compute a static over-approximation of *sdep* and *tdep*. For the purposes of this chapter, we assume that the dependence relations *sdep* and *tdep* are available. In practice the *precise* relations are uncomputable and a reasonable over-approximation is required.

6.4. Divergence Analysis Algorithm

We show the Divergence Analysis algorithm in Listing 1 implemented as a simple worklist algorithm. It operates on the abstract state as given by Definition 19.

Listing 1: *compute_da*. Compute vector shapes for all variables in the program.

```
Input: $InitShapes$ : Initial vector shapes provided by the user.
Output: $Shapes$ : Computed vector shapes for all variables in the program.
// Initial state (Definition 19).
$Shapes = \{\, x \rightarrow \bot \mid x \in id \cup tmn \,\}$ // All variables and terminators mapped to ⊥.
$sdep^* = \{\ \}$
$tdep^* = \{\ \}$
$adep^* = \{\ \}$
$cdep^* = \{\ \}$
// Apply predefined shapes.
foreach $(I, InitShape) \in InitShapes$ do
    $update\_shape(I, InitShape)$
end
// Initialize work queue.
foreach $I \in Region$ do
    $Queue \leftarrow I$
end
while $I \leftarrow Queue$ do
    if $I \in InitShapes$ then continue;
    // Query abstract transformer (see Section 6.5).
    $VarAllocas, NewShape = compute\_transformer(I)$
    // Stack-object divergence (see Section 6.6).
    foreach $VaryingAlloca \in VarAllocas$ do
        $update\_shape(VaryingAlloca, \mathtt{v})$
    end
    // Control-divergent terminator I.
    if $is\_terminator(I)$ and $NewShape \neq \mathtt{u}$ then
        $prop\_control\_div(Block(I))$ // See Listing 3.
    end
    // Push all users of I.
    $update\_shape(I, NewShape)$
end
```

Definition 19. *(Divergence Analysis State) The state of the divergence analysis (the "x" in the fixpoint iteration) is defined by:*

- *Shapes - a mapping from variables (id) and terminators to vector shapes;*
- *$sdep^* \subseteq \mathtt{V}$ - blocks known to be sync dependent on a divergent terminator.*
- *$tdep^* \subseteq \mathtt{V}$ - blocks known to be temporal dependent on a divergent terminator.*
- *$adep^* \subseteq \mathtt{V}$ - blocks known to have a varying activation mask.*
- *$cdep^* \subseteq \mathtt{V}$ - blocks known to have a varying control mask.*

Walkthrough. The DA starts off with a predefined mapping from variables to vector shapes, *InitShapes*. In Whole-Function Vectorization, these are the shapes of the parameters of the function. In Outer-Loop

Vectorization, *InitShapes* provides shapes for the recurrences of the loop to vectorize, e.g. iteration variables.

After initializing the worklist with all instructions and setting the predefined shapes, the procedure enters the worklist loop. The procedure in Listing 2 serves two ends throughout the DA: First, it enqueues all users of the just updated variable. Second, when invoked with a *varying* shape for a variable that is used as block predicate, the procedure enqueues all instructions that are affected by it.

Listing 2: *update_shape*. Update mapped shape of I and enqueue dependent operations.

```
Input: $I$ : Instruction or terminator, $NewShape$ : Vector shape to assign to $I$.
Data: $Shapes$ : Current vector shape mapping, $Queue$ : The work queue.
if $NewShape = Shapes(I)$ then return;
// Enqueue data dependent users of I.
foreach $User \in Users(I)$ do
    $Queue \leftarrow User$
end
// Activation dependence (varying block predicate).
if $NewShape \neq \mathrm{u}$ then
    foreach $b \in blocks\_with\_predicate(I)$ do
        $adep^* \leftarrow b$
        foreach $I' \in Insts(b)$ do
            $Queue \leftarrow I'$
        end
    end
end
$Shapes(I) \leftarrow NewShape$
```

The abstract transfer functions for instructions and terminators are called in Line 14 of Listing 1. The transfer functions query the DA state by the names introduced in Definition 19. In case `I` is an instruction, the transformer computes a new vector shape for the result variable of `I`. In case of a terminator, the transformer returns a non-uniform vector shape if the terminator was found to be control-divergent.

The abstract transformers also return the set *VarAllocas*. This set contains the base pointers of stack allocations through `alloca`, which become non-uniform through `I`. We discuss the uniformity analysis of stack objects in detail in Section 6.6. For now, we assume that *VarAllocas* is an empty set, which is the case e.g. for all arithmetic operations. Finally, when a terminator is found to be divergent (Line 18), the procedure in Listing 3 exercises the induced divergence effects.

The procedure *prop_control_div* in Listing 3 is called whenever the terminator of a block was detected to cause control-divergence (block *DivBlock*). The procedure obtains sync and temporal dependences in Line 1 using the algorithm presented in Chapter 7. Line 26 updates the set of blocks that control-depend on a divergent terminator.

Listing 3: *prop_control_div*. Enqueue operations affected by control divergence in *DivBlock*.

```
Input: $DivBlock$ : Block with divergent terminator.
Data: $sdep^*$ : Known sync divergent blocks, $tdep^*$ : Known temporal divergent blocks, $adep^*$ :
      Known activation divergent blocks, $cdep^*$ : Known blocks with a divergent control mask.
// Compute sync and temporal dependence relation (described in Chapter 7).
$tdeps, sdeps = global\_joins(Successors(DivBlock), \{\ \}, (lp\ DivBlock))$
// Sync dependence.
foreach $sdepBlock \in sdeps$ do
    if $sdepBlock \in sdep^*$ then continue;
    $sdep^* \leftarrow sdepBlock$
    foreach $\phi \in sdepBlock$ do
        $Queue \leftarrow \phi$
    end
    // AllocaSSA (see Section 6.6).
    foreach $allocaJoin \in allocaJoins(sdepBlock)$ do
        $Queue \leftarrow allocaJoin$
    end
end
// Temporal dependence.
foreach $sdepBlock \in sdeps$ do
    if $tdepBlock \in tdep^*$ then continue;
    $tdep^* \leftarrow tdepBlock$
    foreach $\phi \in tdepBlock$ do
        $Queue \leftarrow \phi$
    end
    // AllocaSSA (see Section 6.6).
    foreach $allocaJoin \in allocaJoins(tdepBlock)$ do
        $Queue \leftarrow allocaJoin$
    end
end
// Activation dependence (divergent control mask).
foreach $cdepBlock \in cdep\_on(DivBlock)$ do
    if $cdepBlock \in cdep^*$ then continue;
    $adep^* \leftarrow cdepBlock$
    $cdep^* \leftarrow cdepBlock$
    foreach $I \in cdepBlock$ do
        $Queue \leftarrow I$
    end
end
```

6.5. Abstract Transformers

The divergence analysis builds on a family of transfer functions to model the abstract effects of instructions and terminators (Line 14 of 1) The transfer functions of all instructions have the following form.

$$\mathit{VarAllocas}, \mathit{NewShape} \leftarrow \mathit{compute\_transformer}(\dots)$$

All abstract transformers may access the state as given by Definition 19. The rules in Figure 6.5 provide access to the vector shapes of operands, which can either be variables (id) or constants (literals in the program text). The vector shape *NewShape* is the resulting vector shape of the variable that the operation assigns to (if any). For **store** operations, *VarAllocas* contains all stack allocations ($id =$ `alloca`...) that may diverge as a result (see Section 6.5.2).

$$\frac{}{[\![id]\!]^\sharp = \mathit{Shapes}(id)}\ \text{[DA-ID]} \qquad \frac{c \text{ is a constant}}{[\![c]\!]^\sharp = \mathrm{u}}\ \text{[DA-CONST]}$$

Figure 6.5.: Basic transformers for variables (id) and constants.

[DA-DARITH] and [DA-UARITH] apply to side-effect free operations, such as arithmetic. The special activation query function (**mask**) is handled by [DA-UMASK] and [DA-DMASK]. These rules are shown in Figure 6.6. We cover ϕ node transformers in Section 6.5.1 and memory accesses in Section 6.5.2.

The analysis is intra-procedural. We leave it to the user to supplement abstract transformers for calls to specific functions. If there is no user-supplied transformer, functions without side-effects may be handled by [DA-DARITH] and [DA-UARITH].

$$\frac{\mathit{Block}(id) \in \mathit{adep}^*}{[\![id = \ \mathbf{mask}()]\!]^\sharp = \emptyset, \mathrm{v}}\ \text{[DA-DMASK]} \qquad \frac{\mathit{Block}(id) \notin \mathit{adep}^*}{[\![id = \ \mathbf{mask}()]\!]^\sharp = \emptyset, \mathrm{u}}\ \text{[DA-UMASK]}$$

$$\frac{\exists i. \left(\mathrm{v} = [\![val_i]\!]^\sharp\right)}{[\![id = op\ \overline{val_i}]\!]^\sharp = \emptyset, \mathrm{v}}\ \text{[DA-DARITH]} \qquad \frac{\forall i. \left(\mathrm{u} = [\![val_i]\!]^\sharp\right)}{[\![id = op\ \overline{val_i}]\!]^\sharp = \emptyset, \mathrm{u}}\ \text{[DA-UARITH]}$$

Figure 6.6.: Generic transfer functions for the activation query and side-effect free instructions (arithmetic, ...).

6.5.1. ϕ Nodes

Figure 6.7 shows the defining rules for the abstract transformer for ϕ nodes. The rule [DA-PHI-D] takes care of control-induced temporal divergence in LCSSA ϕ nodes (due to $tdep^*$) and sync divergence for regular ϕ nodes (due to $sdep^*$). In absence of sync or temporal divergence, the result shape of a ϕ node simply depends on its operands, as defined by [DA-PHI-U].

$$\frac{\mathit{Block}(id) \in \mathit{sdep}^* \cup \mathit{tdep}^*}{[\![id = \phi\ \overline{[val_j, \ell_j]}]\!]^\sharp = \emptyset, \mathrm{V}}\ \text{[DA-PHI-D]}$$

$$\frac{\mathit{Block}(id) \notin \mathit{sdep}^* \cup \mathit{tdep}^* \qquad id^\sharp = \bigsqcup [\![val_j]\!]^\sharp}{[\![id = \phi\ \overline{[val_j, \ell_j]}]\!]^\sharp = \emptyset, id^\sharp}\ \text{[DA-PHI-U]}$$

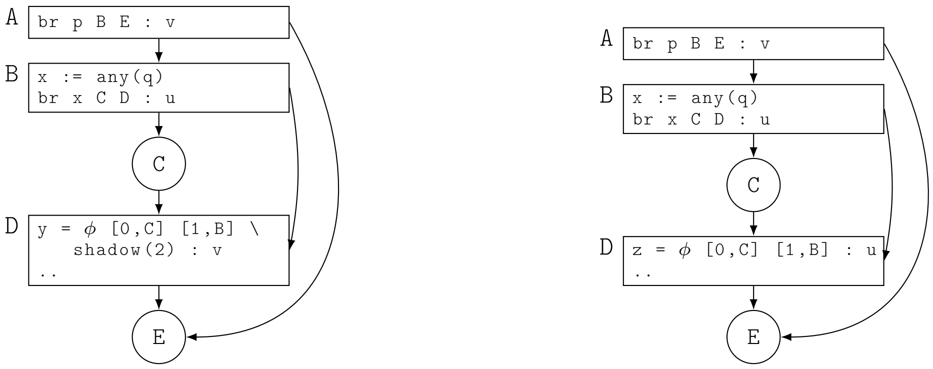
Figure 6.7.: Abstract transformers for non-shadow ϕ nodes.

Shadow ϕ nodes are special in that they have a defined value even when the control mask is *false*. They have the abstract transformer shown in Figure 6.8. The rules for shadow ϕ nodes deviate from non-shadow ϕ nodes in case of a varying activation due to control divergence. Under that circumstance, the shadow ϕ node selects the value of the shadow operand. The example in Figure 6.9 shows how control divergence leads to different outcomes in non-shadow ϕ nodes than in shadow ϕ nodes.

$$\frac{\mathit{Block}(id) \in \mathit{cdep}^* \cup \mathit{sdep}^* \cup \mathit{tdep}^*}{[\![id = \phi\ \overline{[val_j, \ell_j]}\ \mathbf{shadow}(val_s)]\!]^\sharp = \emptyset, \mathrm{V}}\ \text{[DA-SPHI-D]}$$

$$\frac{\mathit{Block}(id) \notin \mathit{cdep}^* \cup \mathit{sdep}^* \cup \mathit{tdep}^* \qquad id^\sharp = \bigsqcup [\![val_j]\!]^\sharp}{[\![id = \phi\ \overline{[val_j, \ell_j]}\ \mathbf{shadow}(val_s)]\!]^\sharp = \emptyset, id^\sharp}\ \text{[DA-SPHI-U]}$$

Figure 6.8.: Abtract transformer for shadow ϕ nodes.



(a) All passive threads in D see the shadow input 2. Variable y is varying despite only uniform sync dependences.

(b) z is uniform and holds either 0 or 1 for all active threads.

Figure 6.9.: In shadow ϕ nodes (y in Figure 6.9a), all active threads see the regular incoming value and all passive threads see the shadow input. All active threads in D reach from either C or B. Since there are no predicates, all threads that are passive in D took the branch from A to E.

6.5.2. Memory Accesses

The transformer rules in Figure 6.10 deal with the divergence analysis of stack objects. Otherwise, since the contents of memory are generally unknown, loads from different addresses are assumed to yield *varying* values.

$\mathit{aliased\_allocas}(val_{ptr})$ returns the set of `alloca` instructions that may alias with val_{ptr}.

$$\frac{P = \mathit{aliased\_allocas}(val_{ptr}) \qquad [\![val_{data}]\!]^{\sharp} = \mathrm{v} \lor [\![val_{ptr}]\!]^{\sharp} = \mathrm{v} \lor [\![pmsk]\!]^{\sharp} = \mathrm{v}}{[\![\,\textbf{store}\ val_{ptr}\ val_{data}]\!]^{\sharp} = P, \bot}\ \text{[DA-STORE-D]}$$

$$\frac{[\![val_{ptr}]\!]^{\sharp} = \mathrm{v}}{[\![\,\textbf{load}\ val_{ptr}]\!]^{\sharp} = \emptyset, \mathrm{v}}\ \text{[DA-LOAD-D]}$$

$$\frac{[\![val_{data}]\!]^{\sharp} = \mathrm{u} \qquad [\![val_{ptr}]\!]^{\sharp} = \mathrm{u} \qquad [\![pmsk]\!]^{\sharp} = \mathrm{u}}{[\![\,\textbf{store}\ val_{ptr}\ val_{data}]\!]^{\sharp} = \emptyset, \bot}\ \text{[DA-STORE-U]}$$

$$\frac{[\![val_{ptr}]\!]^{\sharp} = \mathrm{u}}{[\![\,\textbf{load}\ val_{ptr}]\!]^{\sharp} = \emptyset, \mathrm{u}}\ \text{[DA-LOAD-U]}$$

Figure 6.10.: Abstract transformers for loads and stores.

Consider the three snippets in Figure 6.11. In Figure 6.11a, the *varying* `tid` is stored to `A`, which is a stack allocation. Hence, by [DA-STORE-D], the stack-allocated object `A` is varying as each thread sees a different value stored to it.

However, stack-object divergence also occurs when storing uniform values. In Figure 6.11b, only the constant 42 is written to the stack object `B`. Yet, since each thread stores the value to a different offset `B[tid]` the state of the stack objects still diverges.

Finally, in Figure 6.11c a uniform value is written at a uniform offset into `B`. Hence, the object `B` itself remains uniform and values loaded from the identical offset, here `z`, are uniform as well.

```
A = alloca i64 : v
store A tid
z = load A : v
```

(a) Storing a non-u value.

```
B = alloca i64[64] : v
// ..
bp = elemptr B tid
store bp 42
// ..
bpp = elemptr B 3
z = load bpp : v
```

(b) Storing at different offsets.

```
// Init:
//  i : u
C = alloca i64[64] : u
cp = elemptr C i
j = mul 2 i
store cp 42 j
// ..
cpp = elemptr C 7 : u
z = load cpp : u
```

(c) Uniform values and offsets.

Figure 6.11.: Stack uniformity (Figure 6.11c, handled by [DA-STORE-U]) and divergence (Figure 6.11b and Figure 6.11a, both handled by [DA-STORE-D]). In all snippets `tid : v` is given.

6.6. Alloca SSA for Divergence Analysis of Stack Objects

When threads in a P-LLVM program allocate stack memory through `alloca`, each thread obtains a private thread-local object. The DA analyzes stack objects specially with the goal of proving that all threads perform the same operations on their thread-local copy. If a stack object is shown to be *uniform* that way, the code generator emits only one scalar allocation that is shared across all threads. This is similar to how scalar variables (e.g. `float`) do not need to be widened if all lanes hold the same value (the variable is *uniform*). However, the performance impact for stack objects is far more pronounced since they involve memory allocations and traffic. This can be reduced for each stack object that is shown to be *uniform.*

We create a state abstraction for stack-allocated objects, reminiscent of SSA for May-Alias Information [Cytron and Gershbein, 1993] or *Memory SSA* [Novillo et al., 2007]. This abstraction allows us to handle control-induced divergence of stack objects with the same logic used for regular value joins (ϕ nodes). The *Alloca SSA* abstraction is build up of three kinds of nodes:

`#x = Def(`val_{alloca}`)`	The uninitialized stack-allocated object.
`#w = Write {#x`$_i$` ..}`	The resulting state of the objects `#x`$_i$ after a write has occurred.
`#a = Join {#x`$_i$` ..}`	`#a` joins the definitions from incoming blocks (SSA join point).

The `Join` operator is used in the same places as ϕ nodes to represent value joins.

May-Alias Queries. *aliased_allocas* can also be applied to *AllocaSSA* nodes. In this case *aliased_allocas*(#*x*) returns the set of allocas that may alias with the stack objects represented by `#x`. Those are the allocas occuring in `Def` nodes that reach `#x` in the Alloca SSA graph.

```
1  // v1,v2,v3 : v
2  // u1,u2 : u
3  A:
4    X = alloca f32[1024]      // #x0 = Def(A)
5    Y = alloca f32[1024]      // #x1 = Def(B)
6    if (v1) {
7  B:
8      xp = elemptr X 0
9      store xp v2             // #x2 = Write {#x0}
10   }
11 C:
12                             // #x3 = Join {#x0, #x2}
13   Z = ϕ[X,A] [Y,B]
14   q = elemptr Z 4
15   if (u1) {
16 D:
17     store q u2              // #x4 = Write {#x1,#x3}
18   }
19 E:
20                             // #x5 = Join{#x3, #x4}
21   yp = elemptr 5
22   s =  load yp
23   ret s
24 }
```

Figure 6.12.: A simple AllocaSSA construction example.

Example. Consider the program shown in Figure 6.12. In the code comments, we show the virtual Alloca SSA graph that is constructed for it. There are two `alloca` instructions in the program and each of them is associated with a new `Def` node (Line 4). The **store** in Line 9 can only alias with the alloca `X`. Hence, it is tagged with a `Write` node that only updates the state of `#x0`. The node `#x1` still reflects the latest state of `Y` after the store and remains valid. The block `C` has incoming blocks `A` and `C`. Since the incoming AllocaSSA nodes for `Y` differ, a new `Join` is created (Line 12). The join provides a new Alloca SSA definition for the state of `Y`. The incoming definition from `A` is `#x0` and from `C`, `#x2`, which hence make up the incoming set of the join. Finally, there is a store in Line 17. The pointer variable `q` that is written to may alias with `X` or `Y`. Hence, the write set includes the current valid nodes for both the alloca `X` (node `#x3`) and `Y` (node `#x1`).

6.6.1. Abstract Transformers

$$\frac{\mathit{Block}(\#id) \in \mathit{sdep}^* \cup \mathit{tdep}^* \qquad \mathit{VarAllocas} = \bigcup_{\#n \in S} \mathit{aliased\_allocas}(\#n)}{[\![\#id = \texttt{Join}\ S]\!]^\sharp = \mathit{VarAllocas}, \texttt{v}} \ \text{[DA-JOIN-D]}$$

Figure 6.13.: Abstract transformer for AllocaSSA `Join` nodes.

Use of Alloca SSA in the Analysis. Divergence of stack-allocated objects is induced by **store** instructions in divergent control flow and divergent block predicates. Section 6.5.2 covers the cases for divergent stores. We leverage the AllocaSSA `Join` nodes to handle the control aspect of divergence. If in an abstract way, the `Join` nodes model different incoming states joining from predecessor blocks, just like ϕ nodes do for values. Hence, if a `Join` node in a sync or temporal dependent block may alias with an `alloca` instruction, the thread-local allocated objects may differ. It is hence not surprising that Figure 6.13 closely models the transformers for ϕ nodes, defined in Figure 6.7.

Divergence Effects in the Example. Coming back to the example of Figure 6.12, we can observe how divergence of `X` and `Y` would be detected. The branch condition `v1` is varying and so the DA detects sync divergence at block `C`. The join node `#x3` sits in that block and is put on the queue. By [DA-JOIN-D] (Figure 6.13), all `alloca` that may alias with that join become divergent. Walking back in the Alloca SSA graph, we see that `#x3` only aliases with `X`. Therefore, the vector shape of the alloca `X` turns varying as a result. However, the alloca `Y` is not aliased with `#x3` and remains at its $\perp$vector shape. The branch condition `u1` is uniform and so `E` does not become sync divergent. Finally, the DA infers that `Y` can remain uniform whereas `X` is varying. The code generator may vectorize the allocas as below, assuming that $W = 8$.

```
Xv = alloca f32[8192]  // For thread t the base pointer of X is &Xv[t*1024]
Yv = alloca f32[1024]  // For thread t the base pointer of Y is Yv
```

6.7. Related Work

We discuss the expressiveness of different divergence lattices in Section 10.5. Related work on control-induced divergence is covered in detail in Section 7.5. The comparison in this section looks into the kinds of divergence effects different analyses can detect in principle.

Masked Concretization. The concept of the masked concretizations is foreign to earlier work in the Divergence Analysis domain. This includes *Variance Analysis* by Stratton et al. [2010], which does not consider divergent branches or predicates at all. Works that summon Abstract Interpretation for the definition of the analysis only provide unpredicated concretization functions [Sampaio et al., 2013; Karrenberg, 2015; Alur et al., 2017]. Alternatively, there is an implicit understanding that the vector shapes only apply for actively executing threads [Lee et al., 2013]. We make the notion of masked concretization *explicit* in our divergence analysis using the defined semantics of P-LLVM in the presence of control divergence *and* predication.

Binding-Time Analysis. Aiken and Gay [1998] first noted that binding-time analysis could be used to solve the "single-valuedness" (that is uniformity) problem also for control-flow. Auslander et al. [1996] presents a binding-time analysis where "Stores have no effect on the set of constants". That is stack-allocated objects are not considered.

Non-interference Analysis. Abadi et al. [1999] points out the resemblance of non-interference and binding-time analysis. Several works by Wasserrab et al. [2009]; Hammer and Snelting [2009]; Rodrigues et al. [2016] build on control dependences for non-interference analysis in CFGs. There is a non-coincidental link between dependence in non-inference and activation dependence in the divergence analysis: If the divergence analysis detects that a block is (transitively) control-dependent on a branch condition then its activation becomes varying when the branch condition is varying ([Park and Schlansker, 1991]). P-LLVM function calls (see Section 4.9.2) even take the activation mask as an explicit parameter, whereas non-interference analyses do not consider predication at all.

Whole-Function Vectorizer. The *Vectorization Analysis* by Karrenberg [2015] also analyze stack-allocated objects. However, it ignores the aspect of control-divergence in stores. That is the analysis will fail to detect a control-induced stack divergence, as long as all stores to the stack object store a uniform value at a uniform offset.

Structured Approaches. Alur et al. [2017] present an analysis for uncoalesced memory accesses in CUDA kernels. The analysis is defined for structured control flow. Side-effects of functions are pessimistically over-approximated. However, their analysis is capable of detecting the situation where a block is only ever be executed by a single thread. In that case all divergence effects are mute since there need to be at least two threads in a block for divergence to occur. Since this is achieved by extending the divergence lattice and transformers, this could conceivably be added to our divergence analysis.

The SIMD programming languages, Sierra [Leißa et al., 2014], ISPC [Pharr and Mark, 2012] and Hipacc [Reiche, 2018], all implement divergence analyses for structured control flow only. Among these, ISPC [Pharr and Mark, 2012] is the only to support the divergence analysis of stack-allocated objects. Leißa [2017] present an inter-procedural extension to their divergence analysis algorithm that is call-site polymorphic. Our divergence analysis can be extended similarly by recursively invoking itself on called functions or running as subsolver in an inter-procedural fixpoint loop.

Similar Analyses. The Scalar Evolution analysis [Pop et al., 2005] computes closed-form representations of instructions with regards to their surrounding loops. For example, the loop vectorizer of LLVM uses it to identify uniform instructions. When an operation is not representable in the closed form it is assumed to be varying. It does not detect uniformity for non-polynomial branch conditions or operations with transitive uniformity (for example, branching on a value that was loaded from a uniform offset).

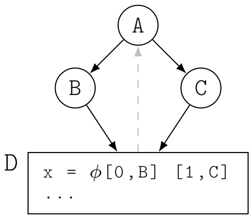
Conclusion. This chapter presents the first divergence analysis for P-LLVM programs. This divergence analysis accounts for all control-divergence effects in P-LLVM programs under the greedy schedule. This includes divergence in the activation mask, sync divergence due to re-converging control and temporal divergence of loops with divergent exits. Earlier approaches for unstructured programs [Karrenberg, 2015] are incorrect in their treatment of the uniformity of stack objects. We resolve this with the use of AllocaSSA (Section 6.6).

Chapter 7.

Static Approximation of Control-Induced Divergence

Divergence in control causes divergence in data, when two threads simultanesouly reach a ϕ node from different predecessors or loop iterations. It is the purpose of the control-divergence analysis to determine which ϕ nodes are affected this way by which branch conditions. The divergence analysis uses this information to compute which variables are uniform, that is always assigned the same value by all active threads, and which are not.

This chapter presents a novel control-divergence analysis for unstructured, reducible CFGs. We prove for DAGs that the algorithm evaluates the underlying disjoint paths criterion precisely and with optimal running time. For reducible CFGs, we show on a set of realistic examples that our algorithm delivers correct and precise results where earlier approaches are less precise or even incorrect.

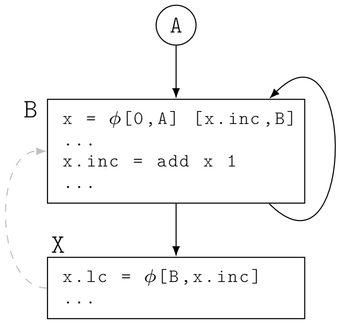


```
x = ϕ[0,B] [1,C]
...
```

(a) `D` is sync dependent on `A`.

ℓ_{sc}	$\ell_{in}{}^{W}$		$\ell_{next}{}^{W\prime}$	
A	⊥	⊥	B	C
B	A	A	$\boxed{\text{D}}$	$\boxed{\text{C}}$
C	B	A	D	D
D	B	C	□	□

(b) Threads part ways at the divergent terminator in `A`. After re-convergence in `D`, divergence is induced: $vec(\mathtt{x}) = \vec{[0,1]}$.



```
x = ϕ[0,A] [x.inc,B]
...
x.inc = add x 1
...
```

```
x.lc = ϕ[B,x.inc]
...
```

(c) `X` is temporally dependent on `B`.

ℓ_{sc}	$\ell_{in}{}^{W}$		$\ell_{next}{}^{W\prime}$	
A	⊥	⊥	B	B
B	A	A	$\boxed{\text{X}}$	$\boxed{\text{B}}$
B	B	B	X	X
X	B	B	□	□

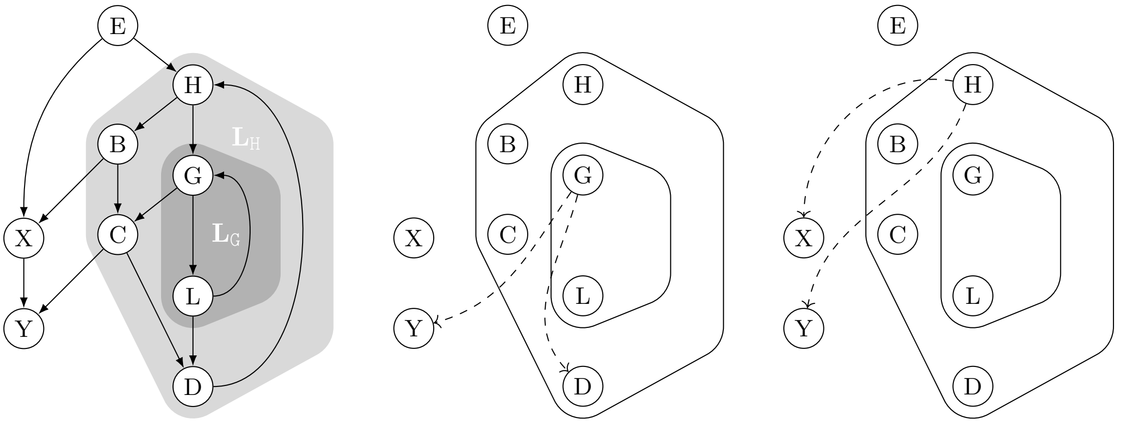
(d) Due to branch divergence in `B`, the first thread leaves the loop after the first iteration, the second after the second. This causes data divergence, $vec(\mathtt{x.lc}) = \vec{[1,2]}$.

Figure 7.1.: **Left:** Sync divergence, **Right:** Temporal divergence. Boxes highlight specific pairs of concurrent next blocks. These pairs are witnesses for the respective control-divergence phenomenon. Gray edges indicate static sync/temporal dependence.

As introduced in Section 6.3, we distinguish two kinds of value divergence induced by divergent control: sync divergence for acyclic control and temporal divergence for loops.

Sync divergence occurs when two threads simultaneously execute the same basic block, coming in from different predecessors. We show an example of sync divergence at runtime on the left of Figure 7.1. Figure 7.1b shows a possible execution of the CFG in Figure 7.1a by two threads. Both threads start out in A, from there the threads part ways. Finally, they reconverge in the block D. By selecting different values for x, the value of x diverges. Therefore, if the terminator in A diverges this may induce divergence in ϕ nodes in D. We say the block D is sync dependent on A.

Temporal divergence occurs when two threads simultaneously reach a loop exit in different loop iterations. On the right of Figure 7.1, we again show a CFG and below an execution of it. The definition of x.inc changes in every loop iteration, it is however the same for all threads in the same iteration. When the first thread exits the loop the value of x.inc for that thread is 1. The second thread takes another iteration and leaves the loop with 2 for x.inc. Despite the fact that threads in the same iteration see the same value for x.inc, its value appears diverged outside the loop. If control flow diverges in a branch this may cause divergence in live-out values (LCSSA ϕ nodes). We say that the loop exit X is temporally dependent on X.



(a) CFG with loop structure. **(b)** Sync dependences of G. **(c)** Temporal dependences of H.

Figure 7.2.: Static sync and temporal dependences in the CFG.

The examples in Figure 7.1 show how sync and temporal dependence can be detected at runtime by witnessing divergent executions. Sync and temporal dependence are dynamic properties in the execution of P-LLVM programs. The Divergence analysis, however, is a static analysis. We thus require compile-time criteria to relate blocks causing control divergence to the affected blocks that become sync and temporal divergent. The technique described in this chapter over-approximates the sync and temporal dependence based on the CFG. We show an example for such static sync and temporal dependence relations in Figure 7.2. Consider Figure 7.2a. Assuming that G is a divergent terminator in the CFG, the resulting sync dependent join points are shown in Figure 7.2b. Figure 7.2c shows the static temporal dependent join points in, assuming control divergence in H.

Sync and temporal divergence is something we observe when a P-LLVM executes. The question is how to approximate sync and temporal dependent blocks with a static analysis that only looks at the CFG and not actual executions. We approach this task with the help of the concept of *Concurrent Next Blocks*. Informally, any two blocks are concurrent next blocks, if at any point in the execution of the program the blocks are next blocks of threads at the same time. The precise semantical definition is given by Definition 20. This is a semantical property. However, we use it to reason why blocks in the CFG can or cannot possibly be concurrent next in any execution.

Definition 20 (Concurrent Next Blocks)**.** *Two blocks $\ell_a, \ell_b \in \mathtt{V}$ with $\ell_a \neq \ell_b$ are called* Concurrent Next Blocks*, written $\{ \ell_a, \ell_b \} \in Cnb$, if there exists a reachable P-LLVM execution state σ_{cnb} with the following property.*

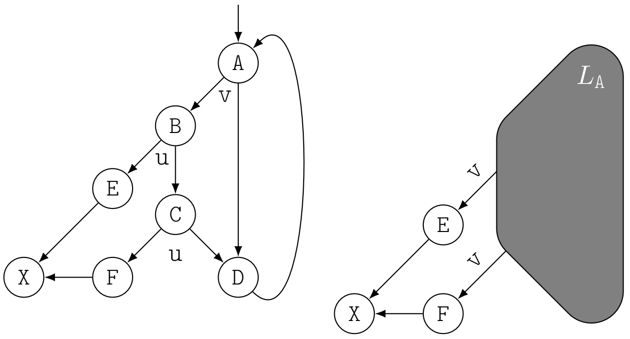
- *At the same time, one threads wants to execute ℓ_a next and another thread ℓ_b. Formally, there exist two threads* $\mathtt{t}, \mathtt{t}' \in \mathcal{T}$ *such that cmsk* $\ell_{in}(\mathtt{t})\ \ell_{next}(\mathtt{t})\ \ell_a$ *and cmsk* $\ell_{in}(\mathtt{t}')\ \ell_{next}(\mathtt{t}')\ \ell_b$.
- *Scheduling either of ℓ_a or ℓ_b complies with the schedule constraints (Chapter 5). That is scheduling either block is synchronizing (Definition 17) and makes progress (Definition 14).*

Note that the set *Cnb* contains only pairs of blocks and that those pairs are unordered, they are sets of size two. In the execution of a P-LLVM program for more than two threads, more than two blocks can be next blocks at the same time. In that case, *Cnb* simply contains the combinations of the next blocks of any two of those threads. Control divergence is characterized by diverged control states, for which two threads are sufficient witnesses regardless of the state of other threads.

We reconsider Figure 7.1 in the light of concurrent next blocks. In each of the execution traces, in Figure 7.1b and Figure 7.1d, a pair of concurrent next blocks is annotated with boxes. The execution traces are witnesses that for the program in Figure 7.1a $\{\,\mathtt{C}, \mathtt{D}\,\} \in$ *Cnb* and that for the program Figure 7.1c $\{\,\mathtt{B}, \mathtt{X}\,\} \in$ *Cnb*. Note that in Figure 7.1a the information that `C` and `B` are concurrent next blocks is enough to anticipate sync divergence at block `D`. This is due to the greedy schedule, which guarantees that the two diverged threads will re-converge at block `D`. Likewise in Figure 7.1c knowing that $\{\,\mathtt{B}, \mathtt{X}\,\} \in$ *Cnb* is sufficient to know that temporal divergence will occur at `X`.

The notion of concurrent next blocks thus decouples sources of divergence, divergent terminators, from sync and temporal divergent blocks that results from the control divergence. This chapter is build on this subdivision. Section 7.1 gives a high-level walkthrough of the top-level algorithm of the control-divergence analysis. For the second part in Section 7.2, we discuss static patterns to identify sync and temporal divergent blocks assuming that an over-approximation of *Cnb* is available. For the third part in Section 7.3, we present a technique to statically over-approximate *Cnb*. Finally, in Section 7.4 we put these two parts together to develop a polynomial time algorithm for static sync and temporal divergent blocks.

7.1. High-level Walkthrough



(a) Initial CFG.

(b) (*lp* A) collapsed into a divergent node.

ℓ_{sc}	ℓ_{next}	$^{W\prime}$
A	B	D
B	$\boxed{\texttt{E}}$	–
D	–	A
A	–	B
B	–	C
C	–	$\boxed{\texttt{F}}$
E	X	–
F	–	X
X	□	□

(c) Schedule with concurrent next blocks at E and F (boxed).

Call	CFG	Detected *sdep*
local_joins { B, D } (*lp* A)	Figure 7.3a	{ D }
local_joins { E, F } (*lp* (*lp* A))	Figure 7.3b	{ X }

(d) Invocations of *local_joins* by *global_joins* for the divergent branch in A.

Figure 7.3.: High-level walkthrough of *global_joins* for a divergent branch in A.

We are looking for an algorithm that given a divergent branch in any reducible CFGs will find all resulting sync and temporal divergent blocks. This is *global_joins*, which we discuss in detail in Section 7.4.2. The algorithm internally calls the more restricted procedure *local_joins*. *local_joins* is restricted in the sense that it will not find sync or temporal divergent blocks if it involves taking a back edge. In order to find all sync and temporal dependent blocks, including those that *local_joins* misses, *global_joins* employs a trick. Conceptually, *global_joins* calls *local_joins* once from the inner to the outer-most loop that surrounds the divergent branch. Each time, it collapses the last loop it visited into a divergent node and considers the loop exits as divergent blocks for the next round. Finally, *global_joins* returns the union of all temporal and sync dependent blocks that the repeated invocations of *local_joins* found.

We illustrate this procedure with the example shown in Figure 7.3. For this example, consider the CFG in Figure 7.3a and further assume that the branch in A is divergent. The execution shown in Figure 7.3c reveals a possible execution where X is sync divergent because of the divergent branch in A.

global_joins takes the following steps, which finally detect X as a sync dependence of A. The algorithm operates on sets of concurrent next blocks. In this case, the block A has a divergent terminator and so initially its successors B and D are possible concurrent next blocks. Figure 7.3d shows the two invocations of *local_joins* that *global_joins* performs. First, *global_joins* calls *local_joins* with B and D as concurrent next blocks. *local_joins* then finds sync dependent blocks that do not require a loop back edge to be taken. In this case, it will find D because if B and D are concurrent next, scheduling B, then C and finally D will make the two threads join in D. This first invocation of *local_joins* does not yield X as it is necessary to schedule the loop header A to get there, which *local_joins* does not consider.

global_joins calls *local_joins* next with the loop exits of inner-most loop that contains A. These are the blocks E and F. In fact, as witnessed in the example protocol in Figure 7.3c, E and F are possible concurrent next blocks. This is conceptually the same as collapsing the loop into a virtual loop node. This is shown in Figure 7.3b with the new divergent node $L_{\texttt{A}}$.

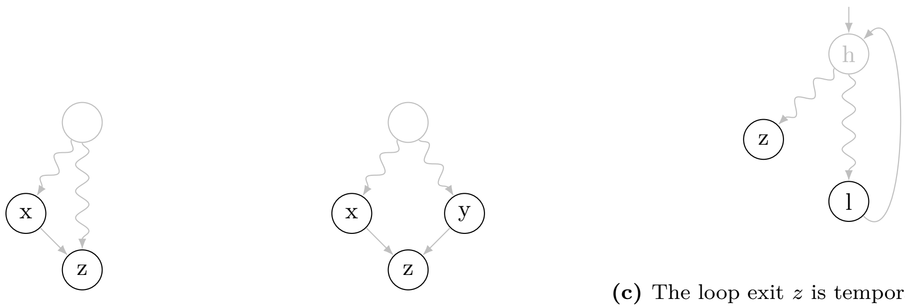
global_joins calls *local_joins* again with `E` and `F` as concurrent next, which finally results in `X` being detected as a sync divergent node.

Since this is the outer-most loop, *global_joins* concludes and returns two things: First, the union of all detected sync dependent blocks, which is the set $\{\,\texttt{D}, \texttt{X}\,\}$. Second, the set of all temporal divergent loop exits that were detected, in this case $\{\,\texttt{E}, \texttt{F}\,\}$.

This concludes the high-level walkthrough. The specifics on how sync and temporal are detected by *local_joins* and *global_joins* will become apparent in the following sections. We continue with the graph patterns that are used to ultimatly identify sync and temporal divergent nodes Section 7.2.

7.2. Static Patterns for Sync and Temporal Divergence

Definition 20 defines Concurrent Next Blocks as a semantic property of a P-LLVM program. We will assume for this section that the set *Cnb* is available. With *Cnb* at hand, sync and temporal divergent blocks are approximated through simple graph patterns. We show the patterns in Figure 7.4.



(a) Sync divergence at z if $\{ x, z \} \in \mathit{Cnb}$.

(b) Sync divergence at z if $\{ x, y \} \in \mathit{Cnb}$.

(c) The loop exit z is temporal divergent if $\{ l, z \} \in \mathit{Cnb}$. The node h is some loop header and l the latch of this loop.

Figure 7.4.: Control divergence patterns that the analysis uses to identify sync and temporal divergent blocks.

The CFG patterns for sync and temporal divergence are formalized in axiom 1 and axiom 2. We assume that these axioms hold true and leave their proof to future work.

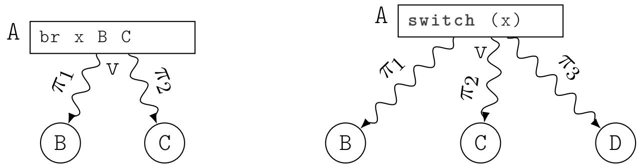
Axiom 1. *(Static Sync Dependence Pattern) A basic block $z \in \mathtt{V}$ is sync dependent on DivBlock $\in \mathtt{V}$, written $z \in sdep(DivBlock)$, if one of the following holds.*

1. *There exists $x \in \mathtt{V}$ such that $\{ x, z \} \in Cnb$ and $x \rightarrow z$. This is the pattern show in Figure 7.4a.*
2. *There exists $x, y \in \mathtt{V}$ such that $\{ x, y \} \in Cnb$ and $x \rightarrow z$ and $y \rightarrow z$. This is the pattern shown in Figure 7.4b.*

Axiom 2. *(Static Temporal Dependence Pattern) A basic block $z \in \mathtt{V}$ is temporal dependent on DivBlock $\in \mathtt{V}$, written $z \in tdep(DivBlock)$, if there is some loop $(lp\ h)$ with DivBlock $\in (lp\ h)$ and $z \in lpexits(lp\ h)$ and $l = lplatch(lp\ h)$ such that $\{ l, z \} \in Cnb$.*

For the sync dependence example in Figure 7.1a, it holds that $\{ \mathtt{D}, \mathtt{C} \} \in Cnb$ and thus we can apply the sync dependence pattern of axiom 1 to obtain that D is sync dependent. Similarly, for the temporal dependence example in Figure 7.1c, the pair $\{ \mathtt{X}, \mathtt{B} \} \in Cnb$. Thus, axiom 2 yields that X is temporal divergent.

7.3. Partial Divergence and Local Concurrent Next Blocks



(a) Local concurrent next at B and C.

(b) Local concurrent next at any pairing of B, C or D.

Figure 7.5.: Given that x is varying, ℓ_{next} may diverge among the threads after block A. If π_i and π_j are almost node-disjoint and do not contain any back edge then their targets are local concurrent next blocks.

If there are almost node-disjoint paths from *DivBlock* to any other two blocks then those blocks may be in *Cnb*. We will call block pairs of this kind *local concurrent next blocks.* Theorem 1 filters pairs of concurrent blocks that are infeasible for greedy schedules (Definition 12).

Theorem 1. *For greedy schedules, a loop header cannot be concurrent next at once with any other block of its loop.*

Proof. We provide a proof sketch: Given any such blocks H and B where H is the header of a loop that contains B. By construction, there has to be a path without back edges from H to B and thus H $\in$ *fwdreaching*(B) (Definition 16). Further assume that H is the next block of a specific thread t. That is ℓ_{next}(t) $=$ H. However, this thread must have actively executed a predecessor of H to proceed to having H as its next block. This can only be the preheader of the loop (say P) or its latch block (L). It cannot have been the latch block because either B is the latch block or B $\in$ *fwdreaching*(L). If B is the latch it cannot have pending threads after it has been scheduled. Further, if B $\in$ *fwdreaching*(L), then the latch cannot have been scheduled because of the constraints of the greedy schedule. Hence, t cannot have reached H from the loop latch. By analogous reasoning, based on greedy schedule constraints, it can also not have been the preheader. This contradicts the initial assumption that $\{$ H, B $\}$ $\in$ *Cnb*. □

However, the two patterns of Figure 7.5 fall short of catching *all* concurrent next blocks. Reconsider the loop in Figure 7.3. There are no two almost node-disjoint paths from A to E and F. Yet, the schedule in Figure 7.3c shows that E and F can be concurrent next blocks.

In the example of Figure 7.3, $\{$ E, F $\}$ is a pair of non-local concurrent next blocks for *DivBlock* $=$ A. The concurrent block pair $\{$ B, D $\}$ is local.

The possible schedule in Figure 7.3c reveals that E is a *divergent loop exit* [Karrenberg and Hack, 2012] of the loop *lp* A. That is in one loop iteration, some threads may take the exit while others continue to the next loop iteration. The block F is a divergent loop exit of *lp* A for the same reason. In effect, while all threads enter the loop at the header, each thread may leave it through a different loop exit.

We summarize the divergence behavior of a loop by considering it as a node on its own, ignoring all internal control flow. The immediate successors of that loop node are its loop exits. For our example, this results in the loop node shown in Figure 7.3b. By abstracting away loop exit divergence into a loop node, we can treat the loop like a block with a divergent terminator. By doing so, the loop blocks E and F are clearly concurrent next; the successor edges to them are trivially almost node-disjoint paths from the virtual loop node.

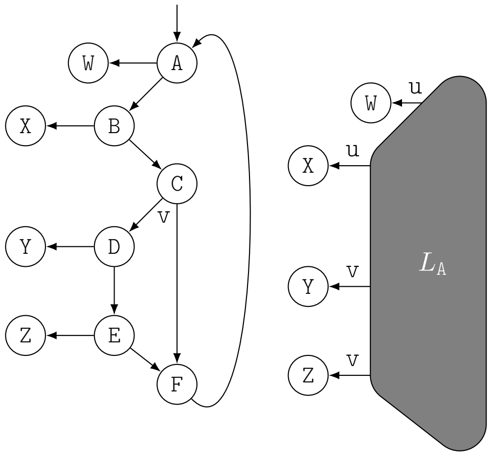
This leads to the following simple scheme for detecting concurrent next blocks: Given a divergent branch that is in a loop, check whether it makes any loop exit divergent. If so, consider the entire loop as one virtual loop node with its loop exits as successors. Recurse on the loop node, treating the loop as a divergent terminator. Along the way collect all local concurrent next blocks.

While this approach to detecting concurrent next blocks works, it is not as precise as it could be on the CFG abstraction. Consider the loop in Figure 7.6, which has two uniform loop exits and two divergent exits. Individual threads may drop out of the divergent loop exits `Y` or `Z` in any iteration, while other threads remain in the loop. However, taking any of the uniform loop exits `W` or `X` will immediately carry all threads out of the loop. In short, the loop exits `W` and `X` are never concurrent next. Yet, by collapsing the loop into a divergent node this information is lost. For the loop node, `W` and `X` will spuriously result as concurrent next blocks.

The two categories uniform and divergent are not precise enough to capture the control divergence out of the exits of divergent loops. To rectify this shortcoming, we introduce the concept of the partially divergent node (Definition 21).

Definition 21 (Partially Divergent Node)**.** *A* partially divergent node $b \in \mathtt{V}$ *is defined by a partition of its successors. Its divergent successor set DivSuccSet and the uniform successor set UniSuccSet. Let $b \rightarrow q$. If $q \in DivSuccSet$, we call the control-flow edge* divergent. *Otherwise $q \in UniSuccSet$ and the edge $b \rightarrow q$ is called* uniform. *The threads reaching a partially divergent node proceed to at most one* uniform *successor but independently to any number of*divergent *successors.*

Reconsidering the divergent loop in Figure 7.6 in light of Definition 21, we observe the following: The loop is modeled as a partially divergent node by treating the uniform loop exits as uniform control-flow edges whereas the divergent loop-exits are turned into divergent control-flow edges.



$\{\,\mathtt{W},\mathtt{Y}\,\} \in \mathit{Cnb}$

$\{\,\mathtt{W},\mathtt{Z}\,\} \in \mathit{Cnb}$

$\{\,\mathtt{X},\mathtt{Y}\,\} \in \mathit{Cnb}$

$\{\,\mathtt{X},\mathtt{Z}\,\} \in \mathit{Cnb}$

$\{\,\mathtt{Y},\mathtt{Z}\,\} \in \mathit{Cnb}$

$\{\,\mathtt{W},\mathtt{X}\,\} \notin \mathit{Cnb}$

(a) *DivBlock* = `C`.

(b) $L_\mathtt{A}$ as a partially divergent node.

(c) Concurrent next blocks due to partial divergence.

Figure 7.6.: Divergent loop as a partially divergent node.

The almost node-disjoint paths criterion needs to be refined to identify concurrent next blocks caused by partial node divergence. We show the resulting concurrent next blocks for the loop node of Figure 7.6b in Figure 7.6c. There are two almost node-disjoint paths from $L_\mathtt{A}$ to the blocks `W` and `X`. Yet, $\{\,\mathtt{W},\mathtt{X}\,\} \notin \mathit{Cnb}$ because `W` and `X` are both uniform successors.

7.4. Computing Temporal and Sync Dependences

Section 7.3 sketches a scheme to over-approximate the set of concurrent next blocks arising from control divergence in *DivBlock*. This section develops that sketch into a procedure to compute all temporal and sync divergent blocks.

Section 7.4.1 starts with two procedures to compute all local results: *local_joins* identifies all local concurrent next blocks and local sync divergent blocks (Listing 4). *local_exits* identifies all local temporal divergent blocks after *local_joins* has run (Listing 6).

Finally, *global_joins* (Listing 7) visits all divergent parent loops of *DivBlock*. In the process, it collects all sync and temporal divergent blocks that are local to *DivBlock* and its divergent parent and ancestor loops. It does so by treating loops with divergent loop exits (temporal divergent exits) as partially divergent nodes.

7.4.1. Computing Local Temporal and Sync Dependences

This section presents a procedure to compute local concurrent next blocks for a given partially divergent node.

The procedure *local_joins* is shown in Listing 4. It starts off from a given partially divergent node with divergent successors *DivSuccSet* and uniform successors *UniSuccSet*. The partially divergent node is considered in the context of its parent loop, *ParentLoop*.

Listing 4: *local_joins*. Identify local concurrent next blocks.

```
Input: $DivSuccSet$ : Divergent successors, $UniSuccSet$ : Uniform successors, $ParentLoop$ :
       Inner-most parent loop.
Output: $SyncBlocks$ : Sync-divergent blocks, $DivLoopExits$ : Temporal-divergent exits from
        $ParentLoop$, $DivLoop$ : Inner-most loop with a divergent exit.
Data: $DomMap \in \mathtt{V} \to (\mathtt{V} \cup \{ \bot, \mathbf{u} \})$ : Disjoint paths map.
$SyncBlocks \leftarrow \emptyset$
// Initialize DomMap
$DomMap \leftarrow \{ b \to \bot \mid b \in \mathtt{V} \}$
foreach $s$ in $DivSuccSet$ do
    $PushDef(s, s)$
end
foreach $s$ in $UniSuccSet$ do
    $PushDef(s, \mathbf{u})$
end
// Compute DomMap
foreach $b$ in $rpo(0, \max(BlockIndex()))$ do
    $d \leftarrow DomMap[b]$
    if $d \neq \bot$ then
        foreach $s$ in $Successors(b)$ do
            $PushDef$(s, d)
        end
    end
end
// Static temporal divergence
$DivLoop, DivLoopExits \leftarrow local\_exits(ParentLoop, ReachedExits)$
```

The main data structure populated by *local_joins* is the *DomMap*, a mapping from blocks to blocks or the special symbol **u**. *local_joins* relies on the *PushDef* procedure (Listing 5) to manipulate the *DomMap*. The *DomMap* symbol **u** marks nodes that are unreachable from forward reaching paths from divergent successors. Theorem 2 shows that *DomMap* directly corresponds to the local concurrent blocks. One application of $PushDef(b, d)$, conceptually updates *DomMap* with the information that there is some control-flow edge into the block b and that d is the *DomMap* value coming in from that predecessor.

Listing 5: *PushDef*. Push control along a control-flow edge to update *DomMap*.

```
Input: b : Block to push definition to, d : Incoming definition.
Data: (all declared in Listing 4) DomMap : Unchanged definition map, SyncBlocks : Set of
      detected join points, ParentLoop : Parent loop from local_joins.
$d' \leftarrow DomMap[b]$
if $d' \neq \bot$ and $d' \neq d$ then
    SyncBlocks insert $b$
    $DomMap[b] \leftarrow b$
else
    $DomMap[b] \leftarrow d$
end
if $b \notin ParentLoop$ then
    ReachedExits insert $b$
end
```

local_joins delivers two byproducts while building the *DomMap*. First, it directly identifies sync divergent blocks using the static sync dependence patterns (Section 7.2). Second, it collects all loop exits of *ParentLoop* that are reachable from the partially divergent node taking only forward edges. Finally, *ReachedExits* is used by *local_exits* to match the static loop dependence pattern.

local_exits of Listing 6 evaluates the static loop dependence criterion in Line 5. At the same time, it determines the inner-most ancestor loop of *ParentLoop* that was left via a divergent edge.

Listing 6: *local_exits*. Detect divergent loop exits.

```
   Input: ParentLoop : the loop.
   Output: DivLoop : inner-most loop containing a divergent exit. DivLoopExits : detected
           divergent loop exits.
   Data: (declared in Listing 4) DomMap : disjoint paths map.
   // Find the inner-most loop whose parent contains a divergent exit.
 1 $DivLoop = \epsilon$ // $\epsilon$ means the virtual top-level loop
 2 if not empty(ReachedExits) then
 3     HeaderDef $\leftarrow DomMap[\text{Header}(ParentLoop)]$
 4     foreach $e \in ReachedExits$ do
 5         if $DomMap[e] \neq HeaderDef$ then
 6             DivLoopExits insert $e$
 7             while $e \in DivLoop$ do
 8                 $DivLoop \leftarrow \ldots$ // child loop of DivLoop towards the divergent node.
 9             end
10         end
11     end
12 end
```

Theorem 2 (Disjoint Path Invariant)**.** *Let DomMap be the result of Listing 9 for a partially divergent node in label A. Consider any two* $p, p' \in \mathbb{V}$. *If* $\mathit{DomMap}[p] = d$ *and* $\mathit{DomMap}[p'] = d'$ *with* $d, d' \neq \bot$, *then the following two statements are equivalent*

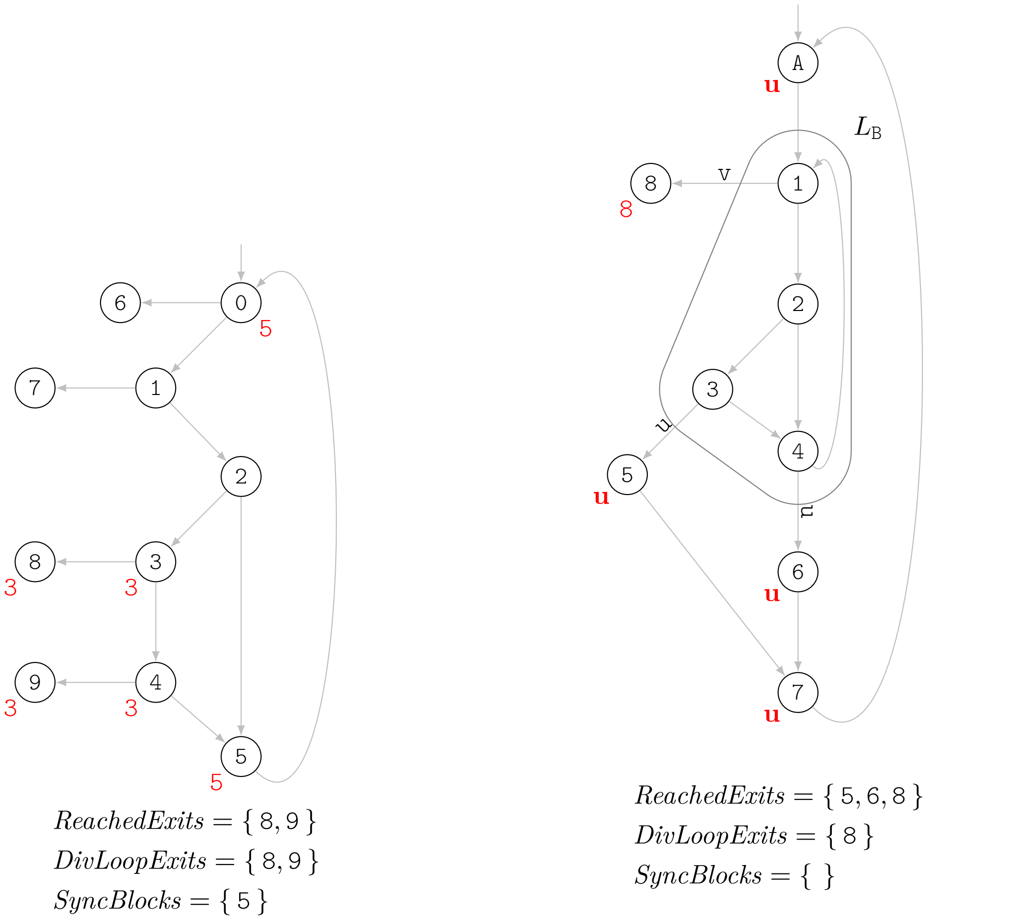
1. $d \neq d'$

2. *There exist almost node-disjoint paths* $t \in A \to q \to^* p, t' \in A \to q' \to^* p'$ *such that at least one of* $A \to q$ *or* $A \to q'$ *is a divergent edge out of A.*

Proof. We provide a proof for the acyclic case and for a variant of *local_joins* in Appendix A. □

According to Theorem 2, if $\mathit{DomMap}[a] = \mathit{DomMap}[b]$, then a and b cannot be local concurrent next blocks. The contraposition ($d \neq d'$) gives us a sound criterion to over-approximate the set of local concurrent blocks.

Examples. We illuminate the inner workings of *local_joins* with two examples: The first, in Figure 7.7a, revisits Figure 7.6a for the case of a divergent terminator in *DivBlock*. The second, in Figure 7.7b, demonstrates the purpose of the **u** symbol on a partially divergent node.



(a) *DivBlock* case ($DivSuccSet = \{3, 5\}$)

(b) Partially divergent loop case ($DivSuccSet = \{8\}$, $UniSuccSet = \{5, 6\}$)

Figure 7.7.: *local_joins* walkthrough. The block labels double as block index numbers. Each red block tags is the blocks *DomMap* entry.

We turn our attention to the example in Figure 7.7a. The example shows the flow of *local_joins* for the divergent block 2. So, 3 and 5, its immediate successors are in *DivSuccSet*. The *ParentLoop* is the inner-most loop containing 2. It comprises the block index range 0 to 5.

After the initialization phase, the *DomMap* only holds entries for 3 and 5, mapping each to themselves. The initialization leaves *ReachedExits* empty as neither of them is a loop exit. The join block is detected in the second rpo phase. When $PushDef(5, 4)$ is called in Line 13, 5 already holds 5 as its *DomMap* entry. Therefore, the dependence pattern of axiom 1 applies here and consequently, *PushDef* will add 5 to the set of *SyncBlocks*. The second phase also reaches 8 and 9 in the traversal and the second phase ends with $ReachedExits = \{8, 9\}$. Finally, *local_exits* evaluates the temporal dependence criterion axiom 2, detecting that $DivLoopExits = \{8, 9\}$. The loop exit 7 is not forward-reachable from 2 and so is confirmed as a uniform loop exit.

The second example in Figure 7.7b covers the scenario of a partially divergent loop. The divergent loop consists of the blocks in the range 1 to 4. The loop has one divergent loop exit, its divergent successor 8 and two uniform exits, its uniform successors 5 and 6. The parent loop of the partially divergent loop includes all blocks except for 8, its only loop exit.

The initialization phase assign to the uniform exits the uniform successor symbol **u**. The divergent loop exit receives 8 and isis immediately added as a reached exit from the parent loop. The threads entering the loop may only proceed to either of 5 or 6. These two blocks are also the only successors of the loop reaching 7. Hence, $\{5, 6\} \notin \mathit{Cnb}$ and 7 is not sync divergent.

In the second phase, during the visit of 5, **u** will be pushed to *DomMap*[7], which was empty before. When the rpo traversal reaches 6, it will also push **u**. Since that is the same symbol as already in the map, 7 is not added as a sync divergent join.

Finally, *local_exits* confirms that 8 is in fact a divergent loop exit also of the parent loop.

Complexity of *local_exits* (Listing 6). Syntactically *local_exits* may appear to have quadratic running time, given its two nested loops.

However, consider the maximum number of iterations that the body of the inner loop, in Line 8, could execute. For each execution of Line 8 of *local_exits*, the variable *DivLoop* is set to a child of the loop that it represents. Further, the loop guard of the inner loop will abort once *DivLoop* is a loop without child loops since no loop exit $e \in \mathit{ReachedExits}$ is part of an inner-most loop.

The set of loops injectively maps to loop headers, which are blocks as well. Hence, the number of iterations of the inner-loop is bounded by the number of blocks for one invocation of *local_exits*. We thus conclude that *local_exits* has $\mathcal{O}(|\mathtt{V}|)$.

Complexity of *local_joins* (Listing 4). The procedure *local_joins* breaks down into three main sections: The first is the initialization phase that visits every successor of the partially divergent node once, hence has a complexity of $\mathcal{O}(|\mathtt{V}|)$. The second phase is the rpo traversal. This takes $\mathcal{O}(|\mathtt{E}| + |\mathtt{V}|)$ time due to the successor loop in *PushDef*. Finally, *local_joins* calls into *local_exits*. As corroborated in Section 7.4.1 *local_exits* also has a complexity of $\mathcal{O}(|\mathtt{V}|)$. In conclusion, *local_joins* takes $\mathcal{O}(|\mathtt{V}| + |\mathtt{E}|)$ to complete.

The existence variant of the two-sources two-sinks node-disjoint paths problem (2-VD) is reducible to *DomMap* queries. In the reduction, *DivSuccSet* contains all sources, pairs of sinks can be checked with the *DomMap* exploiting Theorem 2. Tholey [2012] proves that $\mathcal{O}(|\mathtt{V}| + |\mathtt{E}|)$ is optimal for this instance of the node-disjoint paths problem. Given that the reduction takes constant time, *local_joins* is hence optimal with regards to the *DomMap* computation.

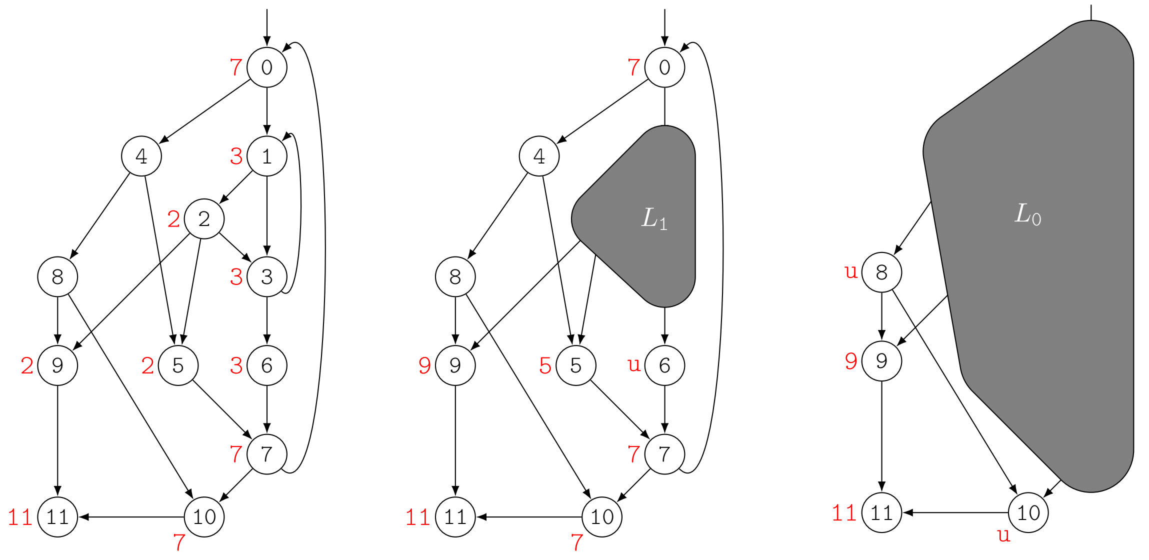
7.4.2. Computing All Temporal and Sync Dependences

global_joins (Listing 7) puts *local_joins* and *local_exits* together to identify all temporal and sync dependent blocks. The sync divergent blocks (`sdeps`) end up in *PathJoins*. The temporal divergent joins (`tdeps`) are found in *LoopJoins*.

Listing 7: *global_joins*. Compute all loop and path splits resulting from a split in x.

```
Input: $DivSuccSet$ : divergent successors, $UniSuccSet$ : uniform successors, $ParentLoop$ :
       parent loop of the partially divergent node.
Output: $LoopJoins$ : divergent loop exits, $PathJoins$ : Reconverging paths join blocks.
Data: $DomMap$ : disjoint paths map.
$LoopJoins \leftarrow \emptyset$, $PathJoins \leftarrow \emptyset$
$DomMap \leftarrow \{\}$
// Listing 4
$SyncBlocks, DivLoopExits, DivLoop \leftarrow local\_joins(DivSuccSet, UniSuccSet, ParentLoop)$
$PathJoins \leftarrow PathJoins \cup SyncBlocks$
$LoopJoins \leftarrow LoopJoins \cup DivLoopExits$
if $DivLoop$ then
    $ParentLoop' \leftarrow lp(DivLoop)$ // The parent loop of $DivLoop$
    $DivSuccSet' \leftarrow Exits(DivLoop) \cap LoopJoins$
    $UniSuccSet' \leftarrow Exits(DivLoop) \setminus DivSuccSet$
    $LoopJoins', PathJoins' \leftarrow global\_joins(DivSuccSet', UniSuccSet')$
    $PathJoins \leftarrow PathJoins \cup PathJoins'$
    $LoopJoins \leftarrow LoopJoins \cup LoopJoins'$
end
```

The algorithm of *global_joins* follows the intuitive scheme of Section 7.1.



$local\_joins(\{\, \mathtt{2}, \mathtt{3} \,\}, \{\,\}, (lp\ \mathtt{1}))$

$SyncBlocks = \{\, \mathtt{3}, \mathtt{7}, \mathtt{11} \,\}$
$DivLoopExits = \{\, \mathtt{5}, \mathtt{9} \,\}$

(a)

$local\_joins(\{\, \mathtt{5}, \mathtt{9} \,\}, \{\, \mathtt{6} \,\}, (lp\ \mathtt{0}))$

$SyncBlocks = \{\, \mathtt{7}, \mathtt{11} \,\}$
$DivLoopExits = \{\, \mathtt{9} \,\}$

(b)

$local\_joins(\{\, \mathtt{9} \,\}, \{\, \mathtt{8}, \mathtt{10} \,\}, \epsilon)$

$SyncBlocks = \{\, \mathtt{9}, \mathtt{11} \,\}$
$DivLoopExits = \{\,\}$

(c)

Figure 7.8.: *global_joins* example. Red labels are the *DomMap* entries for each round of *local_joins*. The block labels double as block index numbers.

Example. Figure 7.8 shows a walkthrough of *global_joins* on a simple loop nest.

In the example, *global_joins* is invoked with $DivBlock = \mathtt{1}$. In the first round of *local_joins*, shown in Figure 7.8a, the partially divergent node is the *DivBlock* itself. Its parent loop contains the blocks from 1 to 3. *DivBlock* has only divergent successors, so any blocks reachable by almost node-disjoint paths from 1 are recognized as *SyncBlocks*. Since there are two divergent exits from the parent loop, *global_joins* goes on to consider the parent loop itself as a partially divergent node, shown in Figure 7.8b The *DivLoopExits* from the last round make up the divergent successors of this loop node. The parent's parent loop is given by the blocks 0 to 7. In the last round, Figure 7.8c, that outer-mode loop is partially divergent node. The loop exits 9 and 10 remain as uniform successors. When *local_joins* visits 8 to call $PushDef(\mathtt{9}, \mathbf{u})$, 9 gets added as a divergent sync to *SyncBlocks*. For the same reason as in shown in Figure 7.7b, the node 10 is proven to not be a sync point. Finally, *global_joins* yields the respective union of the detected sync and temporal divergent blocks.

$$PathJoins = \{\, \mathtt{3}, \mathtt{7}, \mathtt{9}, \mathtt{11} \,\}$$
$$LoopJoins = \{\, \mathtt{5}, \mathtt{9} \,\}$$

Complexity of *global_joins*. *global_joins* ascends in the loop tree starting from an inner-most node. In each iteration *global_joins* uses a finite number of set operations on sets of basic blocks. Each of these operations can be performed in $\mathcal{O}(N)$ time. Let D be the depth of the loop tree (longest chain of nested loops). Combined with the $\mathcal{O}(M+N)$ time for *local_joins*, we conclude that *global_joins* takes $\mathcal{O}((M+N)D)$ time.

7.5. Related Work

Binding-Time Analysis. Aiken and Gay [1998] first noted that that binding-time analysis [Jones et al., 1989] of partial evaluation could be used to detect the uniformity of variables at join points with regards to divergent control flow. Hornof and Noyé [2000] present a binding-time analysis for structured programs, unstructured control (GOTO) is handled by restructuring the program before the analysis. We discuss in Section 12.7 how re-structuring leads to inferior outcomes in vectorization.

Auslander et al. [1996] introduce a technique to detect divergent control-flow joins (non-constant merges) in unstructured Control-Flow with reducible loops. Paths CNF formulas are constructed containing only uniform branch conditions. At merges, the analysis checks whether the paths formulas at predecessors are disjoint. This means testing in their model requires solving an instance of *SAT*, which is NP-complete [Cook, 1971]. The approach proves uniformity of values within loops. The technique can not directly prove that loops are uniform. They resort to unrolling in this case, which is not possible for loops with an unknown, e.g. parametric, loop bound. In consequence, all uses of loop-defined values are pessimistically assumed to be varying (non-constant).

Disjoint Paths Criterion. The *disjoint path criterion* [Havlak, 1993; Karrenberg and Hack, 2012] can be understood as a composition of the criterion for local concurrent next blocks (Definition 20) and the sync dependence pattern of axiom 1. The same can be done for the temporal dependence pattern (axiom 2) yielding the *latch path criterion* [Karrenberg and Hack, 2012]. Karrenberg [2015] defines the disjoint paths criterion for sync dependences but does not present an algorithm to evaluate it. The partially divergent nodes presented in this chapter, summarize the complex divergence effects of loops. That way the known path criteria for divergence are lifted to nested cyclic control flow, which has not been considered in related work.

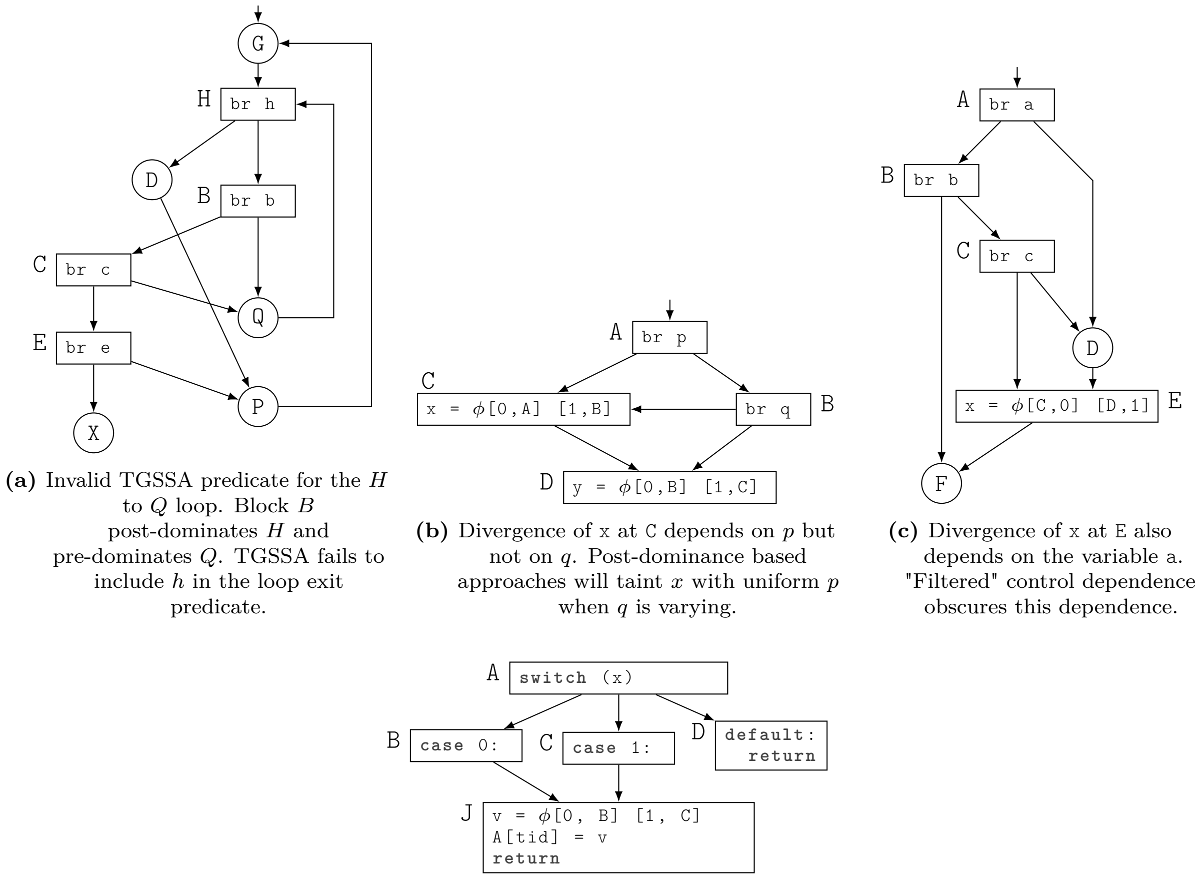
Gated SSA. The family of Gated SSA (GSSA) [Ballance et al., 1990; Havlak, 1993] techniques converts ϕ nodes into dags of select nodes (γ nodes). The leaves are made up of the incoming definitions of the original ϕ node and the branch conditions that effect the ϕ nodes outcome. Gated SSA has been proposed as a solution to the control-induced divergence problem for Divergence Analysis [Sampaio et al., 2013]. If a branch condition in the γ expression is found to be divergent then so that branch induces join-divergence in the ϕ node.

GSSA produces absolute predicates, these are imprecise on unstructured CFGs. Consider the CFG of Figure 7.9c, the GSSA expression (notation by Ballance et al. [1990]) for x will be $\gamma(a, \gamma(b, \perp, \gamma(c, 0, 1)), 1)$. However, the value of b is irrelevant for selecting among the incoming values of the ϕ node. These unnecessary joins could be removed by post-processing, however at the cost of added complexity. This was fixed in Thinned Gated SSA [Havlak, 1993].

The GSSA techniques that we are aware of [Ballance et al., 1990; Tu and Padua, 1995b] use a single loop exit predicate for *all* exits of a loop. For example, the GSSA loop exit predicate for the H to Q loop is $\gamma(h, 1, \gamma(b, \gamma(c, 1,), 0))$. Thus, it could be inferred that both loop exits D and C become temporal divergent with any of the conditions h, b or c. In contrast, our algorithm produces individual results per loop exit that are more precise. It determines that temporal divergent for D only depends on h and that of E only on c and b.

In the same example Figure 7.9a, Thinned GSSA [Havlak, 1993] fails to construct valid loop predicates - it generates the loop predicate $\gamma(b, \gamma(c, \mathit{true}, \mathit{false}), \mathit{false})$ for the H to Q loop, which does not contain h.

Published GSSA algorithms only consider unconditional and binary branches whereas our algorithm also supports partially divergent nodes and switches, such as in Figure 7.9d.



(a) Invalid TGSSA predicate for the H to Q loop. Block B post-dominates H and pre-dominates Q. TGSSA fails to include h in the loop exit predicate.

(b) Divergence of `x` at `C` depends on p but not on q. Post-dominance based approaches will taint x with uniform p when q is varying.

(c) Divergence of `x` at `E` also depends on the variable `a`. "Filtered" control dependence obscures this dependence.

(d) Divergence of `v` at `J` depends on `x`. This is lost in filtered control dependences. Techniques in related work (GSSA) often do not consider terminators of degree greater than 2 at all.

Figure 7.9.: Example CFGs where earlier analyses fail or are less precise.

Techniques for Structured Control Flow. This category comprises algorithms for structured programming languages (AST based) and algorithms for CFGs that make strong structural assumptions. The compiler for the data-parallel ISPC language [Pharr and Mark, 2012] performs its analysis on the AST.

In the context of GPU kernels, it is often assumed that reconvergence only happens at the immediate post dominator of a divergent branch [Coutinho et al., 2011]. Fully structured vectorization analysis [Reiche, 2018]. These fail on the unstructured CFG in Figure 7.9b. When p is varying, the divergence in x at D goes undetected because reconvergence is only anticipated at F, the immediate post dominator of A. Note that NVIDIA architectures starting from Volta allow re-convergence *before* the immediate post dominator NVIDIA [2017, Fig. 21].

Transitive Control Dependence. For some techniques the set of all transitively control-dependent blocks is considered for sync dependence [Lee et al., 2013; Liang et al., 2016]. In that setting, a block will be considered sync divergent if any of its predecessors is transitively control dependent on a divergent terminator. We will refer to this as absolute uniformity. This is identical to the notion of implicit dependence used in program slicing techniques and also common in non-interference analyses [Wasserrab et al., 2009; Hammer and Snelting, 2009; Rodrigues et al., 2016].

Absolute control-dependence techniques are imprecise for divergence analysis. This shows in all the examples ranging from Figure 7.9a to Figure 7.9d. For example, absolute uniformity will not detect uniform x for uniform c in Figure 7.9c when a is varying.

Note that loop exits may not be control dependent on their temporal dependences (single-exit loop).

Filtered Control Dependence. Absolute control-dependences (set of transitively control-dependent blocks) are sometimes refined by subtracting the control dependences of the join block. This is used by AMD HSAIL (`https://reviews.llvm.org/D50433#1193813`) and also relied upon in published work [Lloyd et al., 2019]. However, these approaches fail on Figure 7.9c as well as Figure 7.9d.

Fixing the IPD Criterion. The approach by Chandrasekhar et al. [2019] considers all join points above the immediate post dominator of a divergent branch as sync dependent. This fixes the immediate post-dominator technique by Coutinho et al. [2011] for the unstructured case. This fixing approach has earlier been suggested by Collange [2011a]. However, this approach is less precise than ours. Consider the unstructured CFG in Figure 7.9b. When p is uniform and q is varying, x will be flagged as varying at D, despite the fact that q has no impact on the divergence of x, there is only one path.

Computing Disjoint Paths. Listing 4 solves the decision version of the k-sources two-sinks variant of the node-disjoint paths problem for DAGs. Tholey [2012] presents an algorithm that solves, among other things, the 2-sources variant of this problem and shows that $O(|V| + |E|)$ is optimal. Therefore, the algorithm presented in Listing 4 is optimal as well.

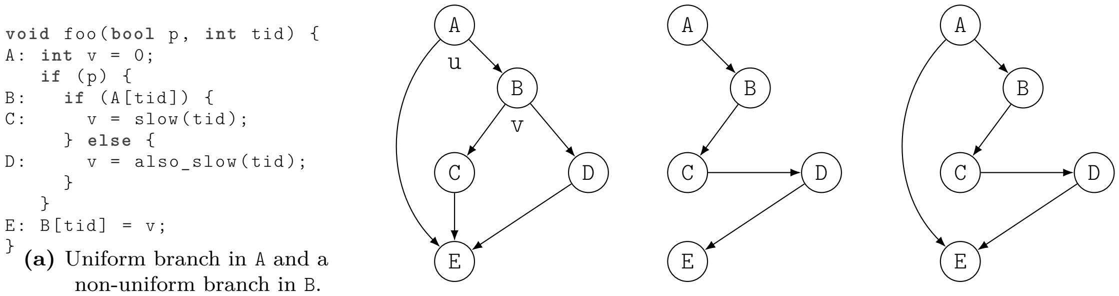
Our technique improves on the existence variant of 2-VD in the following sense: The *DomMap* allows checks in constant time whether there are node-disjoint paths from any pair of nodes from *DivSuccSet* to a given pair of sinks.

Conclusion. We present the first complete, polynomial time algorithm to detect all control-induced joins caused by a divergent terminator in unstructured CFGs with reducible loops. Related techniques are either incomplete, i.e. do not detect all divergence effects, require structured control-flow, e.g. an AST, or have a much higher complexity while achieving a similarly precise result. The concept of partial control divergence is novel for the best of our knowledge. The algorithms presented in this chapter were improved after the research period for this thesis and published [Rosemann et al., 2021].

Chapter 8.

Partial Control-Flow Linearization

Divergence Analysis (Chapter 6) identifies uniform and divergent branches. To generate SIMD code for a CFG, all branches must be uniform, that is branch the same way for all threads. The standard technique for this is if-conversion, which removes all branches, including the uniform ones. This can lead to inefficient code. Figure 8.1a shows a function that calls slow functions in `C` and `D`. If-conversion removes the uniform branch in `A`, as shown in Figure 8.1c. The slow functions calls in `C` and D execute even if p is *false*. This chapter presents partial linearization, which retains the uniform branch in `A`. The partially linearized CFG in Figure 8.1d skips the slow function calls if `p` is *false* leading to faster SIMD code.



```
void foo(bool p, int tid) {
A: int v = 0;
   if (p) {
B:   if (A[tid]) {
C:     v = slow(tid);
     } else {
D:     v = also_slow(tid);
     }
   }
E: B[tid] = v;
}
```

(a) Uniform branch in `A` and a non-uniform branch in `B`.

(b) CFG of 8.1a. **(c)** If-converted. **(d)** Partially linearized.

Figure 8.1.: 8.1a: The branch in `B` is non-uniform (varying) because it depends on the thread id. 8.1c: If-conversion removes all branches. This makes the SIMD code slow as the function calls in `C` and `D` execute even if `p` is *false*. 8.1d: Partial linearization retains the uniform branch in `A`. The slow function calls are skipped if `p` is *false*.

Partial Control-Flow Linearization is a partial if-conversion algorithm for unstructured DAGs. It if-converts all divergent branches while retaining uniform branches. Different to existing techniques, partial linearization does not introduce any new blocks or branches and provides strong guarantees on the control flow retained. The algorithm operates in linear time in the number of edges in the DAG. The resulting DAG is control uniform and trivially vectorizable by replacing every instruction with a SIMD instruction and leaving branches untouched. We show how partial linearization can be used on reducible CFGs with *uniform* loops. All threads that enter a uniform loop leave it in the same iteration and through the same loop exit. The divergent loop transform presented in Chapter 9 transforms non-uniform loops into uniform loops. Partial linearization together with the divergent loop transform can thus make all reducible CFGs control uniform.

8.1. High-level Walkthrough

We begin with the following observation: when a branch in a DAG is if-converted (even if other uniform branches remain), the branch will have only one successor after the transformation and the former successors will post-dominate it. We see this in the initial example. In both transformed DAGs, Figure 8.1c and Figure 8.1d, the blocks `C` and `D` post-dominate block `B`. The key insight behind partial linearization now is that we can sweep through the DAG from top to bottom and draw up new successors for each node such that the existing post-dominance constraints are always satisfied.

Before going into the details of the algorithm, we walk through an example to shed light on how the algorithm achieves this. This walk through is shown in Figure 8.2.

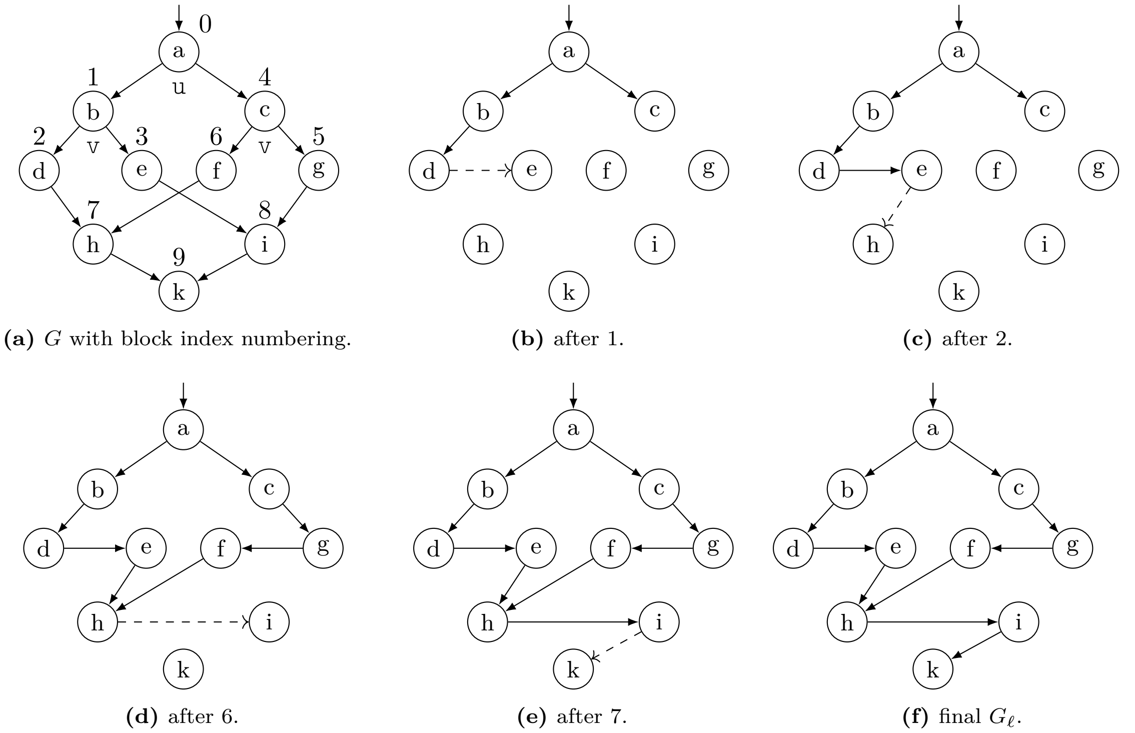
Initially, partial linearization is given a DAG with uniform and divergent branches, as the one in Figure 8.2a. Partial linearization visits every block in order of the block index, a toposort of the nodes that keeps nodes in the same dominance region and loop consecutively together. Partial linearization draws new successors for each node to make sure that divergent branches will only have one successor and all post-dominance constraints are satisfied. The solid edges are the newly generated control-flow edges.

The *deferral relation*, shown as dashed arrows in the figure, keeps track of post-dominance constraints. If $x \to y$ is inserted in the deferral relation, it means that in the final DAG, the block y has to post-dominate x. When partial linearization satisfies a deferral edge, i.e. it establishes post-dominance in the generated DAG, then the deferral edge is removed.

Consider the state after the first two blocks have been processed, shown in Figure 8.2b. Since the block a has a uniform branch, its outgoing control-flow edges are unchanged from the original DAG. The branch in b, however, is varying. This means two things: First, b will have only one successor, in this case d. Second, if d is the new successor then e, the other former successor of b, has to post-dominate d. To enforce this, the algorithm adds the deferral edge $d \to e$ to the deferral relation.

To understand how deferral edges are satisfied, we forward to the point where the next node d has been processed, shown in Figure 8.2c. The node e is drawn as the new successor of d, the reason being that e is the node with the minimum block index number among the successors and deferral targets of d. Since e is now the only successor of d, this satisfies the post-dominance constraint $d \to e$ and the deferral edge is removed. However, we add the deferral edge $e \to h$ to maintain that h has to post-dominate e and by extension d.

Partial linearization proceeds in similar manner through all remaining nodes. We show the intermediate states after visiting f in Figure 8.2d and after the visit of h in Figure 8.2e. The algorithm concludes after k, the last block in the block index, has been processed, shown in Figure 8.2f. The deferral relation is now empty and the set of control-flow edges is final. No divergent branches remain, the uniform branch in a was retained.



(a) *G* with block index numbering.

(b) after 1.

(c) after 2.

(d) after 6.

(e) after 7.

(f) final G_ℓ.

Figure 8.2.: Walkthrough of partial linearization. 8.2a: Source DAG *G* with divergent branches by Karrenberg [2015]. 8.2b to 8.2f: Partially linearized DAG G_ℓ after the specified iteration (block index number). Deferral edges are shown as dashed arrows.

8.2. Algorithm

The partial linearization algorithm works on *loop-free* CFGs. Section 8.3 elaborates how the algorithm can be used for CFGs with reducible loops.

Listing 8: Partial linearization algorithm. Lines 1,4,19 are abbreviations used in the proofs.

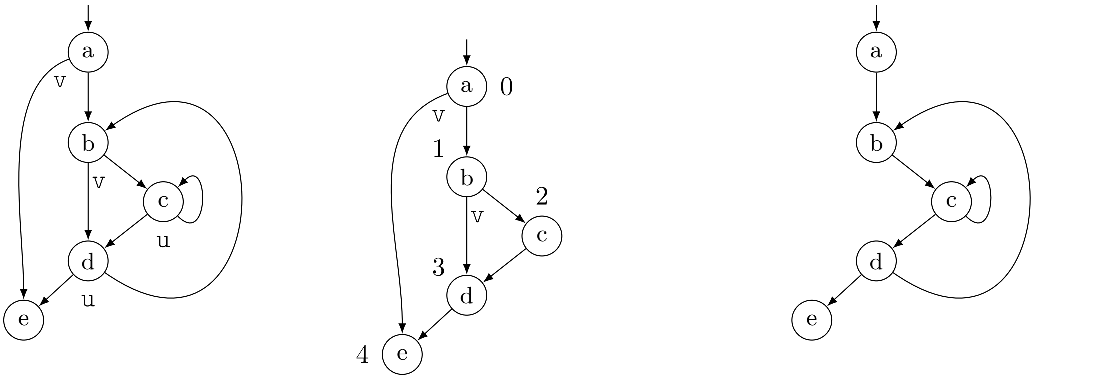
```
   Input: DAG $G = (\mathtt{V}, \mathtt{E}, \mathit{entry})$
   Input: Block index of $G$ (see Section 2.5)
   Output: Partially linearized DAG $G_\ell = (\mathtt{V}, \mathtt{E}_\ell, \mathit{entry})$
 1 // $P \leftarrow \emptyset$
 2 $D \leftarrow \emptyset$
 3 foreach $b$ in $\mathit{BlockIndex}()$ 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             $\mathit{next} \leftarrow \min(T \cup \{s\})$
 9             $\mathtt{E}_\ell \leftarrow \mathtt{E}_\ell \cup \{(b, i, \mathit{next})\}$
10             $D \leftarrow D \cup \{(\mathit{next}, t) \mid t \in (T \cup \{s\}) \setminus \{\mathit{next}\}\}$
11         end
12     else
13         $S \leftarrow \{s \mid \exists i.(b, i, s) \in \mathtt{E}\}$
14         $\mathit{next} \leftarrow \min(T \cup S)$
15         $\mathtt{E}_\ell \leftarrow \mathtt{E}_\ell \cup \{(b, 0, \mathit{next})\}$
16         $D \leftarrow D \cup \{(\mathit{next}, t) \mid t \in (T \cup S) \setminus \{\mathit{next}\}\}$
17     end
18     $D \leftarrow D \setminus \{(b, s) \mid (b, s) \in D\}$
19     // $P \leftarrow P \cup \{b\}$
20 end
```

The *partial linearization* algorithm is shown in Listing 8. The result of the algorithm is a DAG $G_\ell = (\mathtt{V}, \mathtt{E}_\ell, \mathit{entry})$ that constitutes a partially if-converted version of the original DAG $G = (\mathtt{V}, \mathtt{E}, \mathit{entry})$.

The algorithm visits every block in $\mathtt{V}$ in block index order (Line 3) At block b, the algorithm creates outgoing control flow edges from b and adds them to $\mathtt{E}_\ell$, the set of edges in the resulting, if-converted CFG. For divergent branches, a single successor edge is added to $\mathtt{E}_\ell$ in Line 15. If the branch is uniform, one new outgoing edge is added for each successor index of the branch (Line 9).

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 $\mathit{next} \in V$ 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.

The *deferral relation* is given by D. The algorithm ensures that whenever a pair $(v, w) \in \mathtt{V} \times \mathtt{V}$ is put into D, the node w will end up post-dominating v in G_ℓ (Lemma 8 in Appendix B). 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.



(a) G with nested uniform loops.

(b) Backedges are removed, shown with loop compact block index.

(c) G_ℓ with re-inserted backedges.

Figure 8.3.: Handling of loops in partial linearization.

8.3. Partial Linearization of Loops

This section discusses how to extend Listing 8 to support uniform, reducible loop nests. Section 9.1 presents a technique to convert reducible divergent loops into uniform loops. Hence, partial linearization does not have any other restriction than requiring reducible control flow.

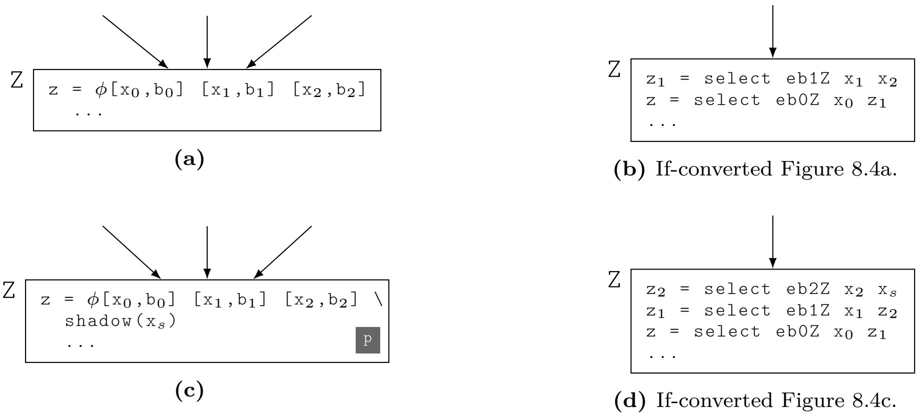
We apply partial linearization to CFGs with uniform loops by deleting loop backedges temporarily. Because we require reducible CFGs, loop headers and back edges can be unambiguously identified. After partial linearization has completed on the DAG, the backedges are re-inserted into G_ℓ. We show an example for this in Figure 8.3.

Running Listing 8 on the CFG that has all backedges deleted is safe because of the following argument: We require the latch block to be unique (Section 2.4). The latch block 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.

8.4. Partial Linearization of ϕ Nodes

As a result of partial linearization, control flow is converted into data flow. When divergent branches are folded down, selection paths in the CFG are removed. This process invalidates the original ϕ nodes as the predecessors of their parent blocks change. Symmetrically to if-converted branches, ϕ nodes have to be if-converted as well.

Consider the ϕ node in Figure 8.4a. It has three incoming edges before partial linearization. Afterwards, there is only a single incoming edge. To preserve the selection behavior of the ϕ nodes in the partially linearized CFG, the ϕ node is lowered into an explicit cascade of `select` instructions. The `select` use predicates to select what was an incoming value in the original ϕ node. We call those *edge predicates* and use the following notation: Whenver `B` executes and `mAB` is true, then in the CFG before if-conversion control reaches `B` from `A`. Turning to the if-converted version in Figure 8.4b, we only require



```
z = ϕ[x0,b0] [x1,b1] [x2,b2]
  ...
```

(a)

```
z1 = select eb1Z x1 x2
z = select eb0Z x0 z1
...
```

(b) If-converted Figure 8.4a.

```
z = ϕ[x0,b0] [x1,b1] [x2,b2] \
  shadow(xs)
  ...
```

(c)

```
z2 = select eb2Z x2 xs
z1 = select eb1Z x1 z2
z = select eb0Z x0 z1
...
```

(d) If-converted Figure 8.4c.

Figure 8.4.: If-conversion of ϕ nodes.

two of the three edge predicates coming into Z. Since this is a non-shadow ϕ node, we can default to the third value if the other two incoming edges are not taken.

For shadow ϕ nodes, such as the one in Figure 8.4c, the shadow input has to be selected whenever the control mask is false. The ϕ node is replaced by the cascade of `select` nodes shown in Figure 8.4d. There is one `select` per incoming value, selecting that value if control reaches the block from this predecessor.

8.5. Correctness

Listing 8 is only concerned with producing a partially linearized CFG and relies on proper predication of the code *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 1. *For each node v that has a predecessor p in P, it holds for v that there is either an edge $(p, b) \in \mathtt{E}_\ell$ or there is another node p' for which there is a path from p to p' in $\mathtt{E}_\ell \cup D$ 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 $\mathtt{E}_\ell \cup D$) 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 $\mathtt{E}_\ell \cup D$ 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 $\mathtt{E}_\ell$, 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. *For each path π of $G = (P \cup F, E)$, there is a path π' in $G_\ell = (V, \mathtt{E}_\ell \cup D)$, 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 1 applies to b.

Consider a path $\pi \in \mathit{entry} \to^* p$ 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 $\pi \circ (p \to b)$ of π to b. By Lemma 1, there is either an edge $(p, b) \in \mathtt{E}_\ell$ or a path $p \to^* b$ in $\mathtt{E}_\ell \cup D$. □

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 this means that any execution trace of the original CFG will also be part of the trace in the partially linearized CFG.

8.6. Preservation of Uniform Control Dependence

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

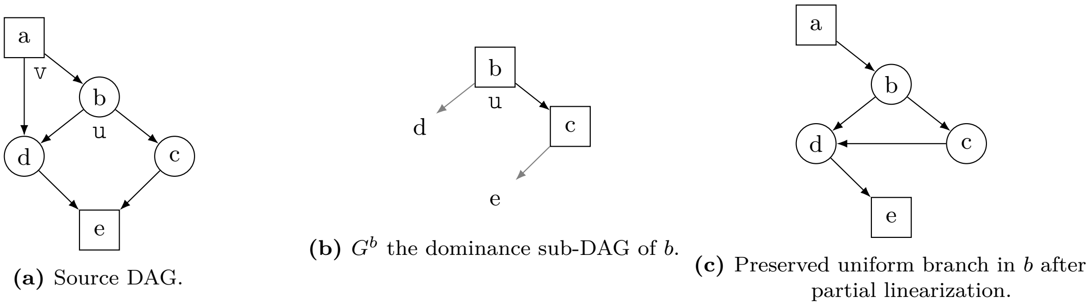
Partial linearization guarantees that predicates can be elided if the predicate of a block is *uniform* even if the predicate is non-constant.

With this guarantee the code generator can safely emit efficient unpredicated instructions for basic blocks with uniform predicates. We make this guarantee precise in Theorem 4 and provide a proof.

Theorem 4. *If $\mathit{uni}(b)$, 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 2, which states that if $\mathit{uni}(k)$ then the control dependences of k are preserved in G_ℓ. We provide the proof for Theorem 4 here and refer the reader to Appendix B for a full technical proof for Lemma 2.

Lemma 2. *If $\mathit{uni}(k)$ then $\mathit{cdep}(k) = \mathit{cdep}_\ell(k)$ where cdep_ℓ is the control dependence in G_ℓ.*



(a) Source DAG.

(b) G^b the dominance sub-DAG of b.

(c) Preserved uniform branch in b after partial linearization.

Figure 8.5.: Preservation of uniform branches in dominance sub-DAGs.

Proof. We now prove Theorem 4. 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 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 $\pi' \in entry \to^* k$ be an arbitrary prefix path to k in G_ℓ. Then, there is an edge $a \to b \in cdep_\ell(k)$ such that $\pi' \in entry \to^* a \to b \to^* k$.

By Lemma 2, $a \to b \in cdep(k)$ 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 $a \to b$ will only be taken in G_ℓ if $a \to b$ is taken in G.

However, $a \to b \in cdep(k)$ implies that $k \succeq^{PD} b$ 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. □

8.7. Preservation of Uniform Branches

Partial linearization preserves uniform branches in blocks with uniform predicates, as implied by Theorem 4. However, the algorithm will even preserve some uniform branches in blocks with varying predicates.

Figure 8.5 shows an example of this. Block b has a uniform branch but its predicate is varying because b is control dependent on the edge $a \to b$, 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 22.

Definition 22. *The dominance region $G^d = (V^d, E^d, d)$ is the subgraph of $G = (V, E, entry)$ that $d \in V$ dominates:*
$E^d = \{x \to y \in \mathtt{E} \mid d \succeq^D x\}$
$V^d = \{x \in V \mid d \succeq^D x \;\vee\; \big(\exists y.y \to x \in E^d\big)\}$

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 23.

Definition 23. *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 8.5, we show the dominance region graph G^d of b in the center. The block b dominates c and so the edge $b \to c$ will be preserved. Generally, as stated by Theorem 5, if an edge $a \to b$ is uniform relative to a node d and d dominates the edge then the edge will be preserved.

Theorem 5. *Given a dominance-compact block index, partial linearization will preserve an edge $b \to y \in \mathtt{E}$ if 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.*

One non-obvious implication of Theorem 5 is that we can insert tests for *all-false* masks in the CFG (BOSCC) [Shin et al., 2009] even before if-conversion (Section 9.2). 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 5 is correct. The full proof can be found in Appendix B. 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 8.6. Block b dominates $b \to d$ and $b \to e$. 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.

8.8. Related Work

Uniform branch preservation has also been studied in the context of GPUs kernels [Lee et al., 2014; Kerr et al., 2012]. 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 [Allen et al., 1983; Baxter and III, 1989].

The *Intel SPMD Program Compiler* (ISPC) [Pharr and Mark, 2012] 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. [2014] showed that

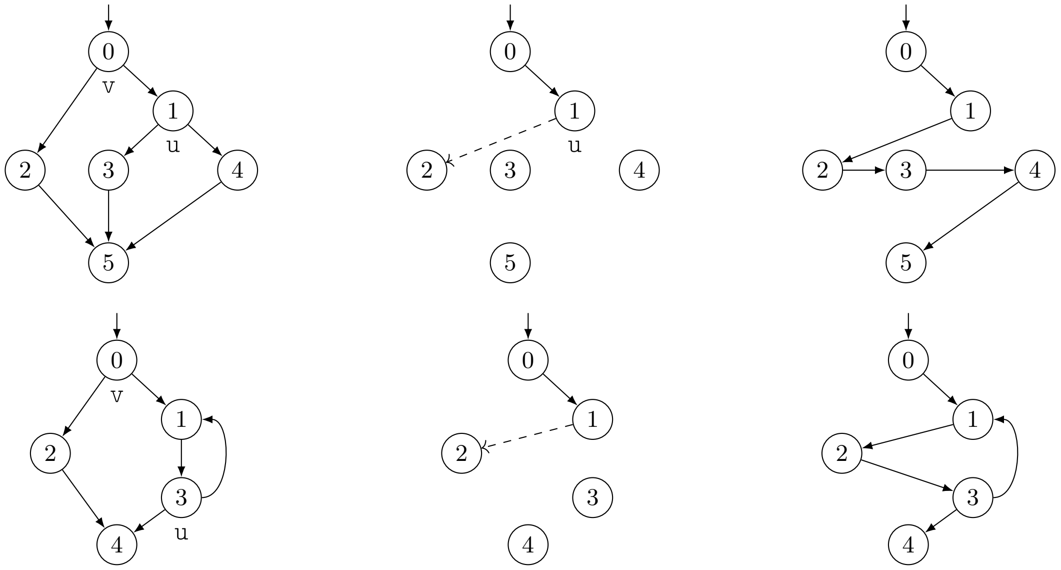
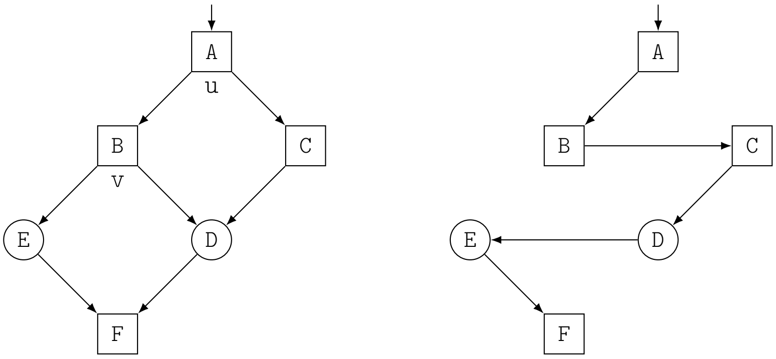


Figure 8.6.: **Top:** Effect of a non dominance-compact block index. **Bottom:** Effect of non loop-compact block index. **Left:** original CFGs G with (non compact) block index, **Center:** processed up to 1, **Right:** G_ℓ with defect.

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 5. Hence, partial linearization is more powerful than ISPC's heuristics.



(a) **(b)**

Figure 8.7.: *Uniform* control dependences indicated with square nodes. 8.7a: Unstructured CFG with uniform predicate at block `C`. 8.7b: Limiting if-conversion to SESE regions will cause `C` to execute even if its predicate is *all false*.

Existing partial if-conversion algorithms do not guarantee that a block with a uniform control mask will only be reached by execution if the control mask is true. Consider the unstructured CFG in Figure 8.7. The block `C` has a *uniform* control mask, yet, after if-conversion it will always execute. Bounding if-conversion to SESE regions, as commonly suggested [Timnat et al., 2014; Lee et al., 2014; Lattuada and Ferrandi, 2017], does not solve this problem: the SESE region of block `C` spans the entire CFG. In short, (bounded) if-conversion does not preserve uniform control dependence.

The early algorithm by Ferrante and Mace [1985] has $\mathcal{O}(n \log n)$ complexity and inserts blocks and branches.

Wahlster [2018] presents a technique to handle divergent control-flow by transforming it into re-converging control flow. Re-converging control-flow is a form of structured control-flow where the immediate post-dominator of a branch has to be a direct successor of the branch. However, no guarantees towards retained branches, synchronization or preserved uniform control dependence are given for that technique.

Karrenberg [2015]; Karrenberg and Hack [2012] present an incomplete partial linearization algorithm that recovers control with additional (so-called *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 8.2. Regarding compile time, partial linearization has linear complexity in the number of edges while the method of Karrenberg [2015] method is quadratic and spans over five algorithm listings. For absence of guarantees the BOSCC-gadget would not reliably work with Karrenberg's method. In fact, the method by Karrenberg [2015] does not preserve uniform control dependence as shown in Karrenberg [2015, Fig. 6.11].

A different class of algorithms insert new basic blocks, predicates and branches after complete if-conversion [Anantpur and Govindarajan, 2014; Shin et al., 2005; August et al., 1999; Warter et al., 1993]. None of the aforementioned techniques gives comparable branch preservation guarantees to partial linearization.

Several techniques have been proposed to enable the loop vectorization of non data-parallel loops [Sampaio et al., 2017; Baghsorkhi et al., 2016]. The techniques presented here are applicable after these techniques have established the legality of vectorization. Techniques such as block unification [Rotem and Ben-Asher, 2014; Coutinho et al., 2011] that improve the utilization in divergent code are complementary to partial linearization.

Chapter 9.

Techniques for Divergent Control-Flow

We present two techniques in this chapter. The divergent loop transform, discussed in Section 9.1, transforms divergent loop into uniform loops. The divergent loop transform is a crucial transformation to make the vectorizer capable of vectorizing divergent control-flow. Second, Section 9.2 presents the BOSCC gadget. The BOSCC gadget is an example for how a known optimization (BOSCC) can be implemented easily exploiting the P-LLVM program representation and the properties of partial linearization proved in Chapter 8.

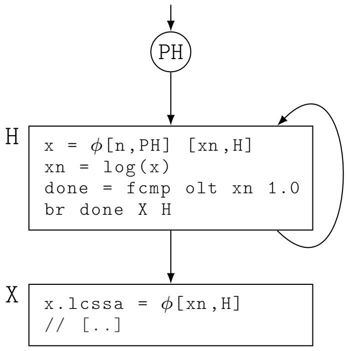
9.1. Divergent Loop Transform

A loop is divergent if threads that enter it may leave it through different loop exits or in a different loop iterations. As discussed in Section 3.6, divergent loops are not directly vectorizable. Partial linearization, presented in Chapter 8, removes control divergence in CFGs with uniform loops. This sections describes the divergent loop transform, which transforms divergent loops info uniform loops.

9.1.1. Procedure

The Divergent Loop Transform operates in two phases. In the first phase, it makes the execution of the loop body conditional on a predicate variable. In the second phase, the control flow towards loop exits is redirected (rebounded) to stay in the loop and ϕ nodes are inserted to keep track of live out values and which loop exits are taken. The final transformed loop is *uniform*.

We will work through the two phases along with the simple divergent loop shown in Figure 9.1.



```
x = φ[n,PH] [xn,H]
xn = log(x)
done = fcmp olt xn 1.0
br done X H
```

```
x.lcssa = φ[xn,H]
// [..]
```

Figure 9.1.: Divergent loop with a single loop exit.

Phase 1: Basic Loop Structure

Figure 9.2a shows the basic loop structure created for the divergent loop in by the first phase. The block `NewH`, `D` and `L` are newly created. The ϕ node `live` is a bitmask that will holds 1 for every thread that executed the loop body and 0 otherwise. The **shadow**(0) operand of `live` guarantees correctness in the presence of divergent control flow outside the loop. Threads that do not steer towards the loop will receive a 0 activation bit in `live`. If no thread remains, `stay` is 0, then control is led to the newly created loop exit `D`. Otherwise, control proceeds to `H` via `Guard`. The purpose of `Guard` is to make execution of `H` conditional on the value of `live`.

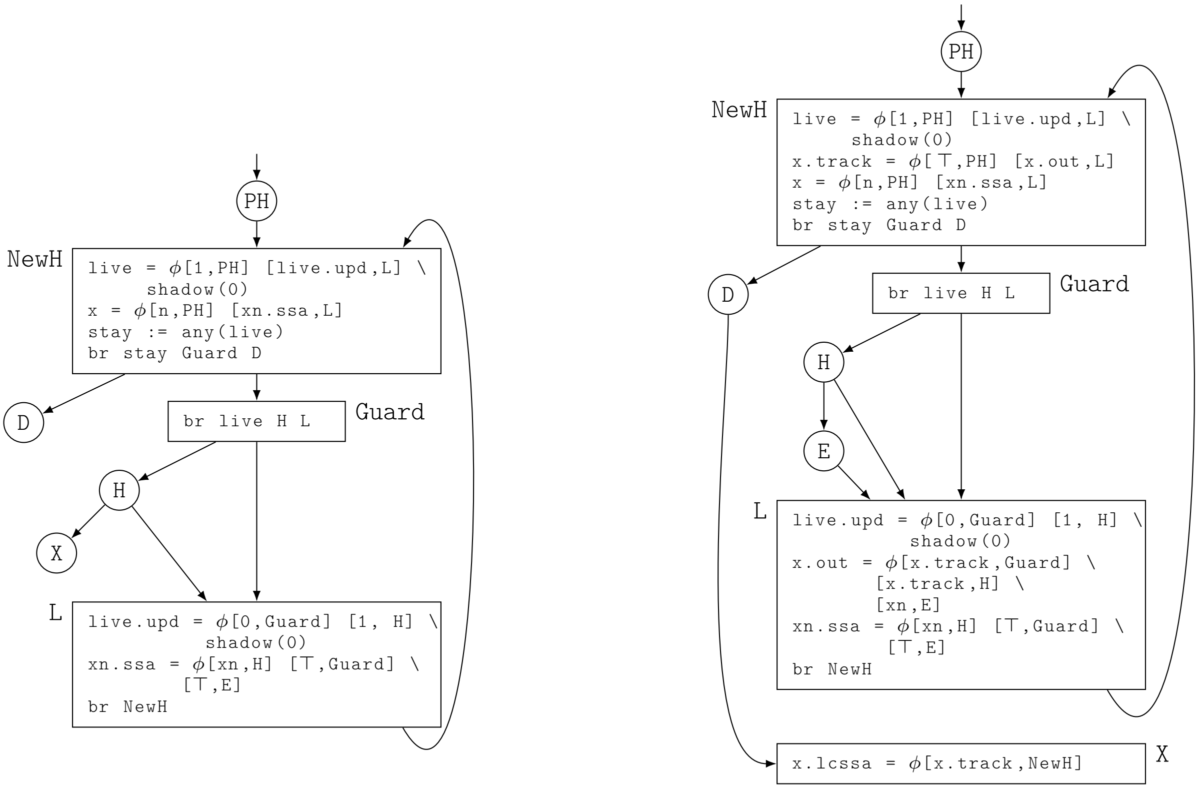
Along with `live`, this phase emits a matching `live.upd` ϕ node in the new latch block `L`. We will later connect new incoming edges to `L` and the `live.upd` ϕ node to change the value of `live` for the next iteration.

Phase 2: Rebounding of Loop Exits

Figure 9.2b shows the result of the second phase. In this phase, every loop exiting edge is redirected, *rebounded*, to the new latch block `L`. In order to allow the ϕ nodes to select based on the incoming edge, the new block `E` is inserted to break the edge. Similar to `live` and `live.upd`, we create pairs of ϕ nodes for every live out value. In the example `xn` lives out of the loop and is used in the exit `X`. The ϕ node x.out records the value that lives out. These values bounce back and forth between x.out and x.track. Eventually, when all threads have taken the E to L edge, the live mask is *all-false*.

The loop header NewH dispatches to the new exit block D. The user x.lcssa of x in E is modified to receive the tracked live out values of x.track.

In case of multiple loop exits, additional tracking infrastructure is added to identify live out threads. We show this in the more complex example in Section 9.1.1.



```
live = ϕ[1,PH] [live.upd,L] \
     shadow(0)
x = ϕ[n,PH] [xn.ssa,L]
stay := any(live)
br stay Guard D
```

```
br live H L
```

```
live.upd = ϕ[0,Guard] [1, H] \
          shadow(0)
xn.ssa = ϕ[xn,H] [⊤,Guard] \
        [⊤,E]
br NewH
```

(a) First phase: Basic loop scaffolding with tracking mask for live thread (live).

```
live = ϕ[1,PH] [live.upd,L] \
     shadow(0)
x.track = ϕ[⊤,PH] [x.out,L]
x = ϕ[n,PH] [xn.ssa,L]
stay := any(live)
br stay Guard D
```

```
br live H L
```

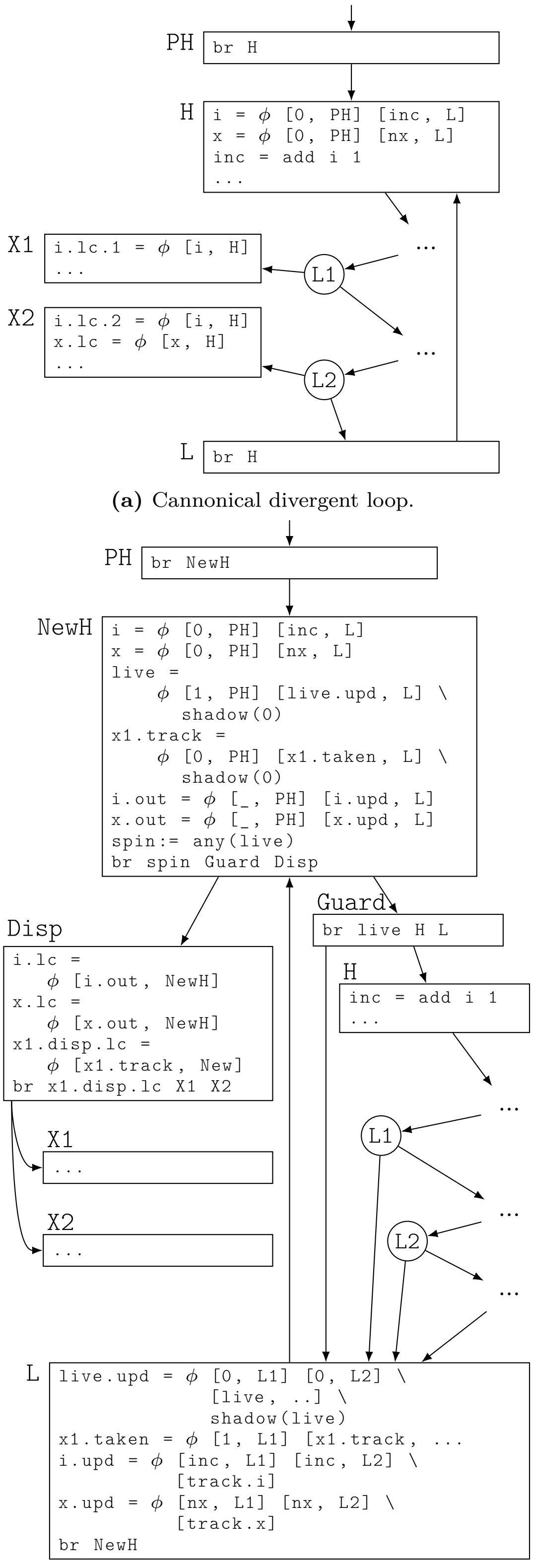
```
live.upd = ϕ[0,Guard] [1, H] \
          shadow(0)
x.out = ϕ[x.track,Guard] \
       [x.track,H] \
       [xn,E]
xn.ssa = ϕ[xn,H] [⊤,Guard] \
        [⊤,E]
br NewH
```

```
x.lcssa = ϕ[x.track,NewH]
```

(b) Second phase: Loop exit X is redirected to stay in the loop.

Figure 9.2.: Two phases of the divergent loop transform.

A Complex Example



```
PH: br H

H:  i = φ [0, PH] [inc, L]
    x = φ [0, PH] [nx, L]
    inc = add i 1
    ...

X1: i.lc.1 = φ [i, H]
    ...

X2: i.lc.2 = φ [i, H]
    x.lc = φ [x, H]
    ...

L:  br H
```

(a) Cannonical divergent loop.

```
PH:   br NewH

NewH: i = φ [0, PH] [inc, L]
      x = φ [0, PH] [nx, L]
      live =
          φ [1, PH] [live.upd, L] \
            shadow(0)
      x1.track =
          φ [0, PH] [x1.taken, L] \
            shadow(0)
      i.out = φ [_, PH] [i.upd, L]
      x.out = φ [_, PH] [x.upd, L]
      spin:= any(live)
      br spin Guard Disp

Guard: br live H L

Disp: i.lc =
        φ [i.out, NewH]
      x.lc =
        φ [x.out, NewH]
      x1.disp.lc =
        φ [x1.track, New]
      br x1.disp.lc X1 X2

H:    inc = add i 1
      ...

X1:   ...

X2:   ...

L:    live.upd = φ [0, L1] [0, L2] \
                 [live, ..] \
                 shadow(live)
      x1.taken = φ [1, L1] [x1.track, ...
      i.upd = φ [inc, L1] [inc, L2] \
              [track.i]
      x.upd = φ [nx, L1] [nx, L2] \
              [track.x]
      br NewH
```

(b) Loop after divergent loop transformation.

We show a more complex application of the divergent loop transform in Figure 9.3a. The loop has two exits, the exiting edge from `L1` to `X1` and the exiting edge from `L2` to `X2`.

Figure 9.3b shows the uniform loop structure that the divergent loop transform will generate for this loop. The transformation succeeds in two steps, a control-conversion step and a data-flow conversion step, detailed below:

In the control-conversion step, the transformation inserts masking infrastructure and re-structures the loop to achieve uniform control-flow. The loop receives a new loop header, `NewH`, that will control the execution of the original loop body. The single live-tracker ϕ node `live` of `NewH` holds the mask of active threads that are designated to execute the loop body in this iteration. The terminator of `NewH` takes all threads out of the loop as soon as all active threads have reached a loop exit - `live`. The new unique exit block `Disp` dispatches control to the former loop exit blocks. The conditional branch in `GuardH` establishes that the former loop body `H`, executes with the explicit live mask `live`.

The original loop of Figure 9.3a is divergent because of its non-uniform loop exits. The divergent loop transform, redirects the loop exiting edges from their former exit destinations to the latch block. This encodes the execution state of each thread in its incoming block to the latch, whether it took a specific loop exit in this iteration of the loop (branch from exiting block; here, `L1` or `L2`), whether the current thread has already left the loop (branch from block `Guard`) or whether the thread will stay in the loop for the next iteration (all other incoming blocks). After this first stage, all divergent loop exits have been removed; the newly inserted loop exit edge from `NewH` to `Disp` is always uniform.

The data-flow stage is concerned with tracking live-out values and installing masking instructions. Newly added ϕ nodes in the header `NewH`, track the live threads of the loop, live-out values and loop exit masks. Each live-out ϕ node, `<id>.out`, holds the value of the live-out variable `<id>` as an original loop exit was taken. The exit tracker ϕ nodes, `<exit_block>.track` in the example, hold the live mask of the loop exit blocks. The transformation inserts a matching set of ϕ nodes in the latch block `L` to update these header ϕ nodes as necessary.

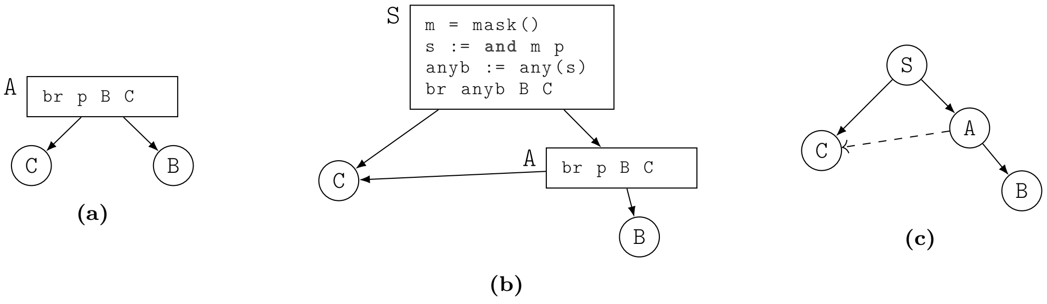
9.2. Branch on Superword Condition Code (BOSCC)

Branch on Supercondition Code (BOSCC) [Shin, 2007] 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 chapter 8, we show that handling BOSCC is contained as a special case in partial linearization by adding a "BOSCC gadget" (see below) to the CFG *before linearization*.

Potential for BOSCC occurs in real benchmarks and applications. Consider the innermost hot loop from 644.nab_s benchmark from SPEC2017 shown in Figure 9.4. The dominating control feature of the loop is a deep if-cascade with very biased branch probabilities, shown as comments in Figure 9.4. 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.

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



```
m = mask()
s := and m p
anyb := any(s)
br anyb B C
```

Figure 9.5.: Basic BOSCC gadget construction. 9.5a: Divergent branch in `A`. 9.5b: BOSCC gadget to skip `B`. 9.5c: Deferral relation at node `A`.

The if-branches in Figure 9.4 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 9.4 leads to a speedup of 9.2% over the Intel C Compiler (icc) on AVX512.

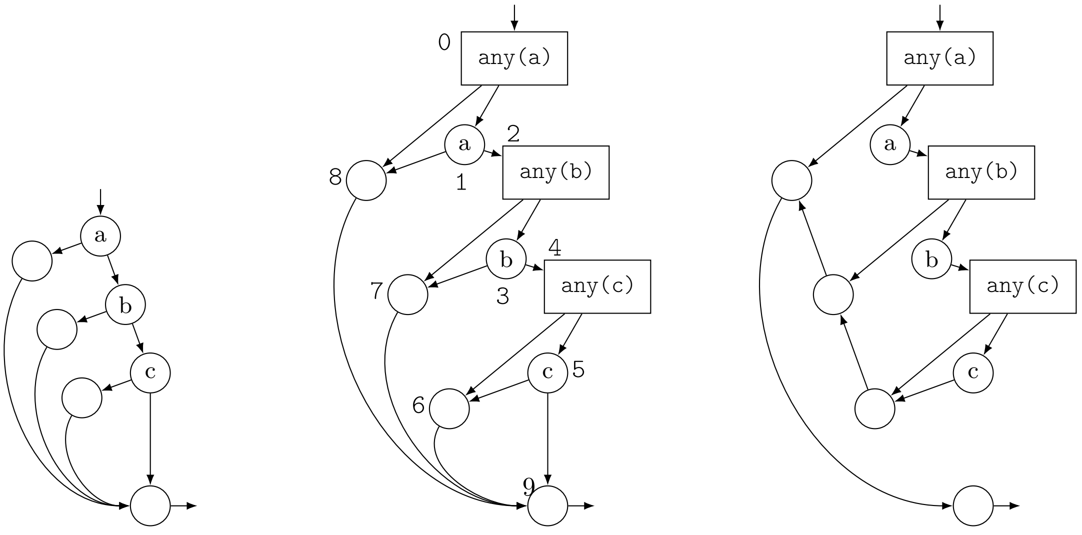


Figure 9.6.: 9.6a: Excerpt CFG from hot loop in nab (Listing 9.4, Line 11 till end). 9.6b: With three nested BOSCC gadgets. 9.6c: After partial linearization.

The BOSCC Gadget

Consider the CFG in Figure 9.5a 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 9.5b 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 9.5c 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 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 9.6, 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 9.6b. Figure 9.6c 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 Figure 9.4. This arrangement lets the linearized CFG skip the remainder of the if-cascade as soon as one of the all-false tests succeeds.

9.3. Related Work

BOSCC gadget. The BOSCC technique [Shin et al., 2009; Shin, 2007] inserts BOSCC branches after if-conversion and requires a predicate hierarchy graph [Mahlke et al., 1992]. The technique presented by Lee et al. [2014] depends on structural analysis and can only insert BOSCC-like tests for SESE regions. 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.

Sun et al. [2019] present a related versioning scheme using *all true* tests. The guarded regions do not require predication since the activation is *all true*. Their implementation is based on the *RV* system and follows the construction of an earlier version of the BOSCC gadget presented in [Moll and Hack, 2018].

The *ISPC* language [Pharr and Mark, 2012] features a *coherent if* statement (cif), which is the equivalent of BOSCC in the structured AST setting. However, the ISPC implementation treats coherent if statements specially, whereas the *BOSCC gadget* is build from standard P-LLVM components: *any* conditions and branches.

Transforming Divergent Loops. Previous work has looked into handling loops with divergent exits [Karrenberg and Hack, 2011; Timnat et al., 2014; Reiche, 2018; Pharr and Mark, 2012; Karrenberg, 2015]. This includes the set up of live masks for divergent loops. Uniform exits in divergent loops were also studied previously [Karrenberg and Hack, 2012]. 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.

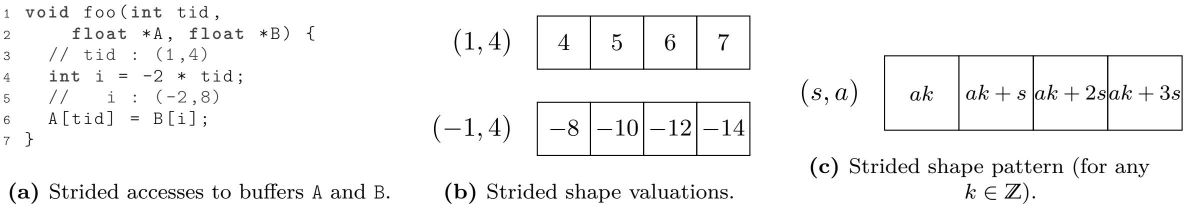
Krzikalla et al. [2016] present a different manual transformation for divergent loops. In their scheme, new work items take the place of iterations that leave a loop early. Lang et al. [2018] present a similar technique in the context of query compilation for databases. There, empty lanes are filled up with new query tuples. We believe that their transformation could be automated on the P-LLVM representation similar to the divergent loop transform.

Chapter 10.

The Stride-Alignment Lattice

This chapter presents the *Stride-Alignment Lattice* (sa-lattice), an abstract analysis lattice for the divergence analysis. The sa-lattice is a refinement of the basic divergence lattice (Section 6.1), which only knows uniform and varying shapes. First, more values and branches can be proven to be uniform as we will show in Section 10.3. Branch conditions that could not be proven uniform by the divergence analysis trigger if-conversion or other mitigation schemes. Unnecessary if-conversion can have a negative impact on the runtime performance of the generated SIMD code. Second, if a value has a strided vector shape, it can be represented by a scalar value in the SIMD code. Third, the widening phase exploits strided shapes for pointers to generate efficient SIMD loads and stores. If a pointer has a strided vector shape that strides in the element size of the pointer, the widening phase can emit an efficient contiguous memory access.

10.1. Abstract Lattice



```
void foo(int tid,
    float *A, float *B) {
  // tid : (1,4)
  int i = -2 * tid;
  //   i : (-2,8)
  A[tid] = B[i];
}
```

(a) Strided accesses to buffers A and B.

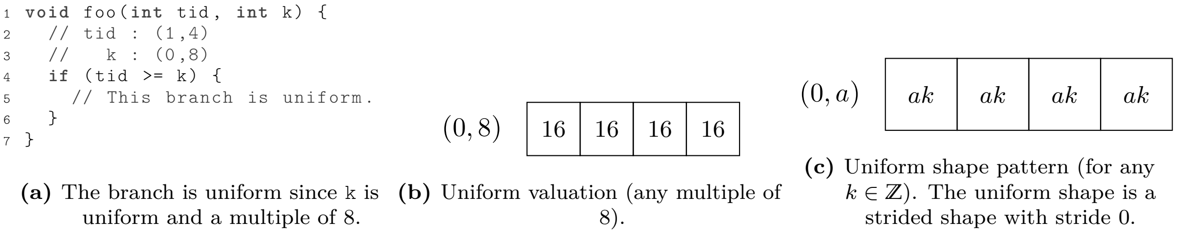
$(1,4)$	4	5	6	7
$(-1,4)$	-8	-10	-12	-14

(b) Strided shape valuations.

(s,a)	ak	$ak+s$	$ak+2s$	$ak+3s$

(c) Strided shape pattern (for any $k \in \mathbb{Z}$).

Figure 10.1.: Strided shapes.



```
void foo(int tid, int k) {
  // tid : (1,4)
  //   k : (0,8)
  if (tid >= k) {
    // This branch is uniform.
  }
}
```

(a) The branch is uniform since k is uniform and a multiple of 8.

$(0,8)$	16	16	16	16

(b) Uniform valuation (any multiple of 8).

$(0,a)$	ak	ak	ak	ak

(c) Uniform shape pattern (for any $k \in \mathbb{Z}$). The uniform shape is a strided shape with stride 0.

Figure 10.2.: Uniform shapes

The sa-lattice distinguishes two classes of shapes, strided and varying, and a bottom element $\perp$ whose instances jointly make up the elements of the lattice. We informally describe the vector shapes as follows.

```
void foo(int tid, int *Idx) {
  int j = 4 * Idx[tid];
  //  j : 4
  v = B[j];
}
```

(a) Varying shape. The lattice retains that j is a multiple of 4.

4	16	−4	0	44

(b) Example for varying valuation.

a	ak_0	ak_1	ak_2	ak_3

(c) Varying shape pattern (for any $k_i \in \mathbb{Z}$). Lanes are unrelated, each of them is a multiple of a.

Figure 10.3.: Varying shapes.

- The class of *Varying* shapes, written a where $a \in \mathbb{N}$ is called the alignment. All of these shapes make no assumption on the relation of threads to each other. However, the value of each thread is a multiple of the alignment (a).
- The class of *Strided* shapes, written (s, a). The value for first thread is a multiple of a, all other threads have the value offset by their thread identifier times the stride s.

Figure 10.1a shows a small example program with two strided memory accesses. The variables `tid` and `i` both have strided shapes. Figure 10.1b shows two possible valuations for four threads for the strided shapes of the example. Since the memory accesses are contiguous, the vectorizer can emit fast strided memory access instructions, which are more efficient than gather or scatter instructions.

In the sa-lattice, uniform shapes are simply strided shapes with a stride of 0. Figure 10.2a shows a branch condition that compares a uniform (stride 0) variable with the strided (stride 1) thread if variable. Juding by the operand shapes, the abstract transformer for `>=` infers that the branch condition is uniform. Figure 10.2b shows a possible valuation for the uniform shape in the example.

These occur, for example, in strided memory accesses. If a memory access known to be strided, the vectorizer emits a SIMD instruction that is more efficient than a gather or scatter.

$$V_{sa} = \{\, \bot \,\} \cup \{\, (s, a) \mid s \in \mathbb{Z}, a \in \mathbb{N} \,\} \cup \{\, a \mid a \in \mathbb{N}, a > 0 \,\}$$

$$\gamma_{sa}(a) = \left\{\, \lambda \mathtt{t} \in \mathcal{T}.ak_{\mathtt{t}} \;\middle|\; k \in \mathbb{Z}^W \,\right\} \qquad \gamma_{sa}(\bot) = \{\ \}$$

$$\gamma_{sa}((s, a)) = \{\, \lambda \mathtt{t}.ak + s\mathtt{t} \mid k \in \mathbb{Z} \,\}$$

Figure 10.4.: Abstract elements (V_{sa}) and concretization (γ_{sa}) for the three classes of vector shapes in the sa-lattice ((s, a) and a and $\bot$).

Figure 10.4 formally introduces the elements of the sa-lattice and their concretization functions. Revisiting the introductory examples, we show the pattern of the concretization functions in the right-most subfigures: Figure 10.1c, Figure 10.2c and Figure 10.3c. In strided shapes, the first lane is always a multiple of the alignment a, all other lanes derive their value from the first lane by adding their lane position times the stride s. In varying shapes, each lane is an independent multiple of a, therefore there is an independent k_i for each lane position i.

The refined shapes of the sa-lattice enable the divergence analysis to prove strided memory access patterns across obscured address arithmetic. Figure 10.5 shows an example for this. Figure 10.5a contains a short code listing with two memory accesses.

We presume that the base pointer `A` is uniform and that `i` has the shape $(1, 8)$. The shape $(1, 8)$ tells us that `i` is contiguous and for the first thread of the thread array, the value of `i` is a multiple of 8.

```
r = A[2*i     ];
s = A[2*i + 1 ];
// p0 == &A[2*i];
// p1 == &A[2*i+1];
```

(a) Source code.

```
x = mul 2 i      : (2,16)
j = add x 1      : (2,1)
p0 = elemptr A x : (16,8)
p1 = elemptr A j : (16,8)
```

(b) Natural IR lowering.

```
x = shl i 1      : (2,16)
j = or x 1       : (2,1)
p0 = elemptr A x : (16,8)
p1 = elemptr A j : (16,8)
```

(c) Address arithmetic obscured by LLVM transformations.

Figure 10.5.: The combination of stride and alignment allows the sa-lattice to prove strided-ness across obscured offset computations. The type of A is f64* and the pointer is aligned to 8 byte.

This shape occurs, for example, if the variable i is the iteration variable of a loop that is vectorized for $W = 8$.

Figure 10.5b shows the P-LLVM IR for the code snippet in Figure 10.5a. The multiplication by the constant factor 2 only leads to a scaling of the stride and alignment of the shape of i. Adding 1 to the shape $(1, 8)$, results in an unaligned value. However, the information that the stride is 2 is preserved through the add. Finally, the address computations defining the pointers p0 and p1 again only rescale the strided offsets into the array A. The strided shapes enabled the divergence analysis to prove that both pointers are strided.

The optimization passes of LLVM (InstSimplify, InstCombine), replace arithmetic instructions by cheaper bit arithmetic. Figure 10.5c shows Figure 10.5b after these passes have replaced the mul with a shl (left shift) and add with an or. Still, exploiting the stride and alignment information contained in the sa-lattice shapes, the divergence analysis is able to prove that p0 and p1 are strided. The shl is conceptually a multiplication by two and again handled by rescaling the strided shape of i. As the alignment of x is 16 it is in particular a power of two and so the least-significant bit of x is always zero. Hence, the or behaves like an addition.

We use the V_{sa} lattice for all pointers and integer values in the program. All other values, such as those of boolean and floating-point type, still use the basic divergence lattice (Section 6.1), distinguishing only uniform (u) and varying (v) shapes.

Machine Integer Interpretation. The sa-lattice is defined in terms of integers and natural numbers. Actual IR uses IR integer types with a finite number of bits. The concretization γ_{sa} is to be interpreted in the native type of the IR value. Let us say that v $: (a, b)$ and the type of v is iN with N being the number of bits. Then γ_{sa} should be interpreted in the native bit width of the integer with the alignment and stride recast to the native integer type, truncating them where necessary. This is possible since the operations in the definition of γ_{sa} in Figure 10.4 are sign-agnostic (multiplication and addition).

If the integer value y is a bit string, we denote by $[y]_S \in \mathbb{Z}$ the signed integer interpretation and by $[y]_U \in \mathbb{N}$ the unsigned integer interpretation.

10.2. Lattice Structure

The divergence analysis presented in Chapter 6 is a fixed-point solver for the abstract divergence analysis lattice. Assuming that all abstract transformers are monotonic and all ascending chains in the divergence lattice are finite, the divergence analysis converges to a fixed point in a finite number of iterations [Kam and Ullman, 1977].

The partial order relation $\sqsubseteq$ of the sa-lattice is shown in Figure 10.6.

$$\begin{array}{rcllll}
\bot & \sqsubseteq & x & & & \forall x \in V_{sa} \\
a & \sqsubseteq & a' & \iff & a'|a & \forall a, a' \in V_{sa} \\
(s,a) & \sqsubseteq & (s,a') & \iff & \max(a',1)|a & \forall (s,a),(s,a') \in V_{sa} \\
(s,a) & \sqsubseteq & a' & \iff & a'|\gcd(s,a) & \forall (s,a), a' \in V_{sa}
\end{array}$$

Figure 10.6.: Partial order of V_{sa} lattice elements (We define $\gcd(a,0) = \gcd(0,a) = 0$).

The basic divergence lattice (Section 6.1) only knows three shapes: varying, uniform and $\bot$. All ascending chains in the basic divergence lattice are trivially finite. It is not obvious that all ascending chains in the sa-lattice are finite because the lattice contains a countably infinite number of vector shapes. The proof for Theorem 6 shows that indeed all ascending chains in the partial order $\sqsubseteq$ of the sa-lattice are finite.

Theorem 6 (Finite Asending Chains). *The partial order $\sqsubseteq$ of the sa-lattice, defined in Figure 10.6, has only finite ascending chains.*

Proof. An ascending chain that starts in a *varying* element stays in the realm of *varying* elements. This is because a *Varying* class shape is never smaller equal than a *Strided* shape in the partial order $\sqsubseteq$. Hence, if the claim is proved for both classes separately it holds for the entire partial order. Since then, any chain that starts in the *Strided* class crosses over to the *Varying* class after a finite number of steps. Any tail chain within the *Varying* class is then again finite. Hence, we only need to show the claim for the cases $a \sqsubseteq a'$ and $(s,a) \sqsubseteq (s,a')$.

In the case of $a \sqsubseteq a'$, the ordering holds whenever $a'|a$. Given that $a \neq a'$, this is only possible if $a = a'k$ for some $k \in \mathbb{N}, k \neq 1$. Therefore always $a > a'$. Since $a \in \mathbb{N}$, on any such chain eventually $a = 1$. Due to the fact that $a \in \mathbb{N}$ this happens after a finite number of steps for any a.
The $(s,a) \sqsubseteq (s,a')$ case follows analogously.

□

10.2.1. Relation to the Basic Divergence Lattice

Most analysis and transformations in the vectorization system only need to know whether branch conditions are uniform or not. For example, Partial linearization (Chapter 8) retains uniform branches and the control-divergence analysis (Chapter 7) will only trigger for non-uniform branch conditions. This fundamental distinction between uniform and non-uniform values is what makes up the basic divergence analysis lattice (Section 6.1).

Some abstract transformers operate on both, the sa-lattice and the basic lattice, at once, e.g. when data is converted. When this happens, shapes of the basic lattice are translated to sa-lattice shapes and vice versa.

Going from the sa-lattice to the basic lattice, translation is done as follows: Every uniform strided shape $(0, a)$ for any alignment a translates to the `u` vector shape of the basic lattice. All other shapes, except for $\perp$, map to `v`, the varying shape of the basic lattice. These are all non-0 strided and varying shapes of the sa-lattice. Going from the basic lattice to the sa-lattice, the basic uniform shape translates to $(0, 1)$, the uniform shape of all uniform integer vectors. The basic varying shape `v` translates to 1, the varying shape of all integer vectors with independent values.

Consider the instruction `z = bitcast f64 x`, if `x` has the type `i64` it has a vector shape of the sa-lattice. The result of the instruction however has the `f64` type and obtains a vector shape of the basic divergence lattice.

We introduce abstract transformers for the sa-lattice for some instructions in Section 10.4. For instructions and cases not included in the sa-lattice transformers, we use the transformers of the basic divergence lattice instead. Where these make use of explicit `u` or `v` in their definition, we adapt the sa-lattice shapes using the equivalences above.

10.3. Example: Comparison to the Basic Lattice

There are cases where the sa-lattice enables the divergence analysis to prove that more values are uniform than possible with the basic divergence lattice. Consider the outer-loop vectorization case shown in Figure 10.7a. Figure 10.7b shows the two results of two runs of the divergence analysis once obtained with the basic lattice (directly after the colon) and once with the sa-lattice (annotated to the right of the basic lattice shapes, separated by a comma).

```
double * A, B;
// ..
#pragma omp simd simd_len(4)
for (i = 0; i < n; ++i) {
  for (j = 4; j < m; j *= 2) {
    A[i / j] = B[j]
  }
}
```

(a) Source code.

```
 1 // i                        : v; (1,4)
 2 H: j = φ [E,4] [B,jN]     : u, (0,4)
 3   e = icmp slt j m        : u, (0,1)
 4   br e B X
 5
 6 B: // j loop body
 7   d = sdiv i j            : v, (0,1)
 8   Aptr = elemptr A d      : v, (0,32)
 9   Bptr = elemptr B j      : u, (0,32)
10   x = load Bptr           : u, (0,1)
11   store Aptr x
12   jN = shl j 1            : v, (0,8)
13   br H
14
15 X: .. // latch of i loop
```

(b) Divergence analysis results with basic lattice shapes (left) and sa-lattice shapes (right).

```
// ..
j.repl = broadcast <4xi32> j
j.v = add <4 x i32> j.repl <0,1,2,3>
d.v = sdiv <4 x i32> i.v j.v

Bptr = elemptr i32* B j
x = load i32 Bptr

Aptr.v = elemptr <4 x i32>* A d.v
x.v = broadcast <4xi32> x
scatter Aptr.v x.v
// ..
```

(c) SIMD code from basic shapes. Basic shapes are insufficient to prove uniformity of the sdiv and store address to the A array. This results in a slow scatter operation.

```
// ..
d = sdiv i32 i j

Aptr = elemptr i32* A d
Bptr = elemptr i32* B j
x = load i32 Aptr




store Aptr x
// ..
```

(d) SIMD code from sa-lattice shapes. Since all shapes are uniform, the generated SIMD code only contains scalar instructions.

Figure 10.7.: Example program annotated with basic lattice shapes and sa-lattice shapes. sa-lattice shapes assuming W divides 4 and A and B are aligned to at least 32 byte.

In Line 7 of Figure 10.7b, the inferred basic lattice shape is varying whereas for the sa-lattice, d is shown to be uniform. The widening phase exploits uniform vector shapes to generate more efficient SIMD code. Using only the basic shapes, the widening phase produces the SIMD code shown in Figure 10.7c. Since the shape of d is varying, the division operation defining d is widened to a vector division. This is worse than it may seem since many SIMD ISAs do not have a SIMD integer division (AVX2). For those targets, the compiler backend splits the SIMD division into four separate scalar division operations. Further, since the pointer Aptr is also varying, a slow scatter operation is generated.

Compare this to the SIMD code that is generated if the divergence analysis is using sa-lattice shapes, shown in Figure 10.7d. Since in fact all inferred sa-lattice shapes are uniform, the SIMD code generator only emits scalar instructions.

10.4. Abstract Transformers

The abstract transformer definitions are partial, that is some of them are not defined if any of their operands have a $\perp$ shape. We lift the abstract transformers to return $\perp$ in that case to obtain definitions for the total input domain. Other unspecified cases defer to the generic transformer of the basic lattice, i.e. [DA-UARITH] and [DA-DARITH] of Figure 6.6. The ϕ nodes and memory accesses are handled by interpreting the definitions of their transformers (provided in Section 6.5.1 and Section 6.5.2) in the sa-lattice domain.

The analysis state is given as a mapping from variables to vector shapes, denoted A. This is a shorthand for the mapping *Shapes* of the divergence analysis state given in Definition 19. The look up of the vector shape of variable x in the current analysis state is given by $A(\texttt{x})$.

10.4.1. Constants

If c is some integer or pointer constant in the program, then it has uniform shape with the literal integer value as its alignment.

$$[\![\mathsf{c}]\!]^\sharp = (0, |\mathsf{c}|)$$

For the corner case $\mathsf{c} = 0$, this results in the $(0, 0)$ shape, which precisely concretizes to the zero vector. We We define the function $\mathit{constval}(x)$ to evaluate to the value of x if x is a constant.

10.4.2. Integer Arithmetic

We define the auxiliary function threadalign in Figure 10.8. The function returns the greatest common alignment of all vector elements that can be inferred from the vector shape.

$$\text{threadalign}(x) = \begin{cases} a & \text{if } x = a \\ \gcd(s, a) & \text{if } x = (s, a) \end{cases}$$

Figure 10.8.: Definition of the per-lane alignment. The function threadalign is not defined on $\perp$.

$$[\![\texttt{add x y}]\!]^\sharp\ A = \begin{cases} (s+s', \mathrm{gcd}(a,a')) & A(\mathrm{x}) = (s,a) \wedge A(\mathrm{y}) = (s',a') \\ \mathrm{gcd}(s',a',a) & A(\mathrm{x}) = (s',a') \wedge A(\mathrm{y}) = a \\ \mathrm{gcd}(a,a') & A(\mathrm{x}) = a \wedge A(\mathrm{y}) = a' \\ [\![\texttt{add y x}]\!]^\sharp\ A & \mathrm{otw} \end{cases} \qquad [\![\texttt{sub x y}]\!]^\sharp = [\![\texttt{add x (-y)}]\!]^\sharp$$

$$[\![\texttt{(-y)}]\!]^\sharp\ A = \begin{cases} a & A(\mathrm{y}) = a \\ (-s,a) & A(\mathrm{y}) = (s,a) \end{cases}$$

$$[\![\texttt{mul x y}]\!]^\sharp\ A = \begin{cases} ca & \mathit{constval}(\mathrm{x}) = c \wedge A(\mathrm{y}) = a \\ (cs, ca) & \mathit{constval}(\mathrm{x}) = c \wedge A(\mathrm{y}) = (s,a) \\ (0, aa') & A(\mathrm{x}) = (0,a) \wedge A(\mathrm{y}) = (0,a') \\ aa' & a = \mathrm{threadalign}(A(\mathrm{x})) \wedge a' = \mathrm{threadalign}(A(\mathrm{y})) \end{cases}$$

Figure 10.9.: Integer arithmetic transfer functions for V_{sa}.

$$[\![\texttt{sdiv x y}]\!]^\sharp = \mathrm{div}_S(\mathrm{x},\mathrm{y}) \qquad [\![\texttt{udiv x y}]\!]^\sharp = \mathrm{div}_U(\mathrm{x},\mathrm{y})$$

$$\mathrm{div}_T(\mathrm{x},\mathrm{y})\ A = \begin{cases} (s/c, a/|c|) & (s,a) = A(\mathrm{x}) \wedge [\mathrm{y}]_T = c \in \mathbb{Z} \wedge c|s \wedge c|a \\ a/|c| & a = A(\mathrm{x}) \wedge [\mathrm{y}]_T = c \in \mathbb{Z} \wedge c|a \\ \mathrm{generic}(A(\mathrm{x}), A(\mathrm{y})) & \end{cases}$$

Figure 10.10.: Signed/Unsigned transfer functions for V_{sa} ($[y]_T$ is the signed or unsigned interpretation of the constant y).

10.4.3. Function Calls

The sa-lattice represents the lane id function precisely with the $(s, 0)$ corner case.

$$A(\texttt{thread\_id()}) = (1, 0)$$

$$\gamma_{sa}((s, 0)) = \{\ [0, s, .., s(W-1)]\ \}$$

10.4.4. Comparison Instruction

Comparison operators demand special attention since they are potentially branch conditions.

```
1 // W = 4
2 tid = thread_id() : (1,0)
3 //vec(tid) = [0,1,2,3]
4
5 // Given k, l : u
6
7 bool b = 4*l + tid >= 36*k; : u
```

Figure 10.11.: Uniformity of integer comparison (variable `b`) using the sa-lattice.

Figure 10.11 shows an example of boolean variable that can be proven to be uniform in the sa-lattice. The variable `tid` has a precise representation in the lattice. The transformer for `icmp` is then able to prove uniformness.

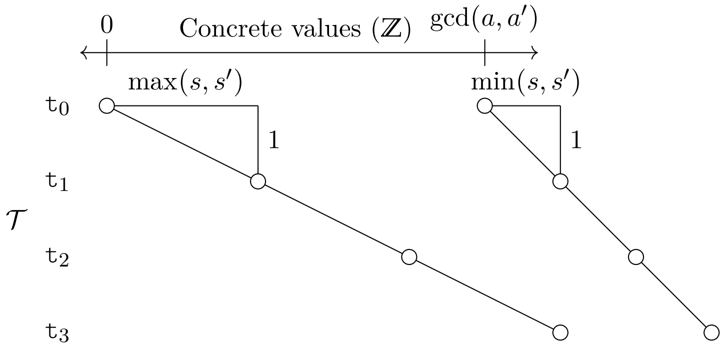
$$[\![\texttt{icmp sge x y}]\!]^\sharp\ A = \begin{cases} \mathrm{u} & \text{if } A(\mathrm{x}) = (s, a) \land A(\mathrm{y}) = (s, a) \\ \mathrm{u} & \text{if } \begin{array}{l} A(\mathrm{x}) = (s, a) \land A(\mathrm{y}) = (s', a') \\ \land \max(s, s')(W-1) < \min(s, s')(W-1) + \gcd(a, a') \end{array} \\ \mathrm{v} & \text{otw} \end{cases}$$

$$[\![\texttt{icmp eq x y}]\!]^\sharp\ A = \begin{cases} \mathrm{u} & \text{if } A(\mathrm{x}) = (s, a) \land A(\mathrm{y}) = (s, a') \\ \mathrm{v} & \text{otw} \end{cases}$$

Figure 10.12.: Abstract sa-lattice transformers for integer comparison.

Figure 10.12 shows the transformer for `icmp sge` (the signed greater-equal comparison) and equality. The other integer comparisons follow an analogous construction and are not presented here. The condition for uniformity of `icmp sge` (signed integer, equal-less-than comparison) with strided operands is visualized in the diagram of Figure 10.13.

The slopes of the two lines are the respective strides of the shapes. This is the case in the signed greater-than-equal comparison in Line 7 of Figure 10.11. Assuming that `k` and `l` have an alignment of 1 (they are unaligned), the vector shape of `4*l + tid` is $(1, 4)$ and for `36*k` it is $(0, 36)$. Therefore, the transformer in Figure 10.12 shows the `b` is uniform.



$$[\![\texttt{icmp sge x y}]\!]^\sharp = \mathrm{u}$$
$$A(\mathrm{x}) = (s, a) \quad \text{and} \quad A(\mathrm{y}) = (s', a').$$

Figure 10.13.: Visualization of the condition for uniformity for `icmp sge` as defined in Figure 10.12. The `y` axis is refers to the thread index (as indicated by the $\mathtt{t}_i$ on the left). Uniformity is detected when the two sloped lines do not cross.

10.4.5. Bit Arithmetic

The encoding of alignment in the vector shapes enables the sa-lattice transformers to detect stride and alignment through certain bit arithmetic operations. We provide here the transformers of `or` and `shl` as examples of the kind of reasoning that is possible with sa-lattice shapes. The `or` and `shl` transformers compute the shown strided shapes for the pointers `p0` and `p1` in Figure 10.5c.

Or. The `or` behaves like an `add` operation for certain operand shapes. Consider the case of $[\![\text{or x } c]\!]^\sharp$ such that c is an integer constant. We shall ignore the case that $c = 0$ since then $A(\mathtt{x})$ is returned. The following applies if there is a $n_c \in \mathbb{N}$ such that the bit $c[n_c - 1]$ is 1 and all bits with higher significance are 0. If further $A(\mathtt{x}) = (s, a)$ and $2^{n_c} | \gcd(s, a)$ then the resulting shape is given by $[\![\mathtt{addx}\ c]\!]^\sharp$.

$$[\![\mathtt{shl\ x\ y}]\!]^\sharp\ A = \begin{cases} [\![\mathtt{mul\ x}\ 2^c]\!]^\sharp\ A & \text{if } \mathit{constval}(\mathtt{y}) = c \\ (0, a2^{a'}) & \text{if } A(\mathtt{x}) = (0, a) \wedge A(\mathtt{y}) = (0, a') \\ a2^{a'} & \text{if } a = \text{threadalign}(A(\mathtt{x})) \wedge a' = \text{threadalign}(A(\mathtt{y})) \end{cases}$$

Figure 10.14.: Left-shift transformer.

Shl. Figure 10.14 shows the transformer for left-shift. The transformer builds on the connection between integer multiplication and the left-shift operation, i.e. $\mathtt{x}2^{\mathtt{y}} = \mathtt{shl\ x\ y}$. When `y` is a constant, the abstract transformer for `shl` defers to the one for multiplication.

10.4.6. Address Computation

The sa-lattice shapes of pointer values are interpreted in the smallest addressable unit of the IR (a byte). The abstract transformer of the `elemptr` instruction hence spells out the offset computation as a expression consisting of `mul` and `add` terms. Those are handled by the abstract transformers for `add` and `mul`, described in Section 10.4.2.

10.5. Related Work

Many earlier works in data-parallel programming languages [Hanrahan and Lawson, 1990] and divergence analyses [Stratton et al., 2010; Yue et al., 2013] conceptually implement the basic divergence lattice. The basic divergence lattice is insufficient to detect contiguous memory accesses since all non-uniform pointers are varying. To this end, an induction variable is sometimes used to detect stridedness in memory accesses [Yue et al., 2013].

The *Whole-Function Vectorizer* by Karrenberg [2015] uses a restricted stride alignment lattice. The lattice encompasses uniform elements, positive unit-strides, and positive strides combined with a binary aligned flag. The consecutive interpretation for pointers is typed, that is a unit-stride in a pointer means that consecutive array elements are addressed whatever the element size may be. The strided class does not identify any particular stride value but represents an unknown stride. Alignment is an on-off property: an integer is aligned, if its a multiple of the vector width, a pointer is aligned if it would justify an aligned SIMD load. The *WFV* lattice does not represent negative strides.

[Sampaio et al., 2013] present the *affine* lattice, which supports stride but not the alignment part of the sa-lattice. Their lattice does not detect the stride in Figure 10.5c. They also discuss general polynomial constraints.

Alur et al. [2017] show which memory instructions are coalesced on in CUDA kernels. That is that the set of addressed elements of a memory instruction is within the range of 12 byte. They introduce an expressive boolean divergence lattice to identify branch conditions that imply that only a single thread reaches a guarded block. To this end, the lattice distinguishes an all-false element (`F`), all-true (`T`), all-true-but-one (`T-`), all-false-but-one (`F-`). It should be noted that they do not consider control-divergence or predication for this property at all. Concerning integer values, they do not model alignment at all but a fixed set of strides $\{ -1, 0, 1 \}$ and varying.

The divergence lattice by Collange [2011a] is very similar to ours. It features (s, a) shapes and the a class. The stride may be $\top$ implying that the stride is not a constant similar to OpenMP [2015] `linear` clauses. However, the alignment is limited to powers-of-two. Abstract transformers are only sketched rudimentary in Collange [2011a, Table 1.]. The transformer for integer multiplication is actually incorrect.

OpenMP [2015] features a divergence lattice similar to the one by Collange [2011a]. There, the alignment is limited to pointers. The lattice is only used to specify function parameters in whole-function vectorization (`#pragma omp declare simd`). For example, the stride may be a loop-invariant value (e.g. a uniform function parameter or an increment `x += p` in loop vectorization). No transformers are provided.

In conclusion, no published divergence analysis presents lattices *and* transformers to detect the strides in Figure 10.5c.

Chapter 11.

Loop-nest Tensorization

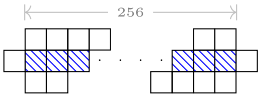
Some codes benefit from considering multiple loops at once in the vectorization process. We refer to the vectorization in multiple nested loops as *tensorization* since a tensor is nothing more than the multi-dimensional version of a vector. This chapter presents the TensorRV system, which generalizes the RV vectorization systems to tensors.

We provide two motivating examples to demonstrate the benefits of tensorization, a 5-point Jacobi stencil and matrix transpose. The tensorized versions in Figure 11.2 and Figure 11.3 were automatically generated from scalar code with our tensorizer prototype.

```
for (j = 1; j < rows - 1; j += 1) {
  for (i = 1; i < cols - 1; i += 256) {
    a = load_v256(A(j-1, i  ));
    b = load_v256(A(j  , i-1));
    c = load_v256(A(j  , i  ));
    d = load_v256(A(j  , i+1));
    e = load_v256(A(j+1, i  ));

    store_v256(B(j,i), .2 * (a+b+c+d+e));
  }
}
```

(a) Vectorized in the `i` loop (width 256).



(b) Cells indicate loaded elements. Hatched cells indicate computed results (a store). In average, 3 unique cells have to be loaded to compute one result.

Figure 11.1.: Vectorized 5-point Jacobi stencil.

The 5-point Jacobi example shows how tensorization naturally results in vector register tiling and vector load coalescing. We show the vectorized version in Figure 11.1 and the tensorized version in Figure 11.2. The loop-vectorized Jacobi kernel in Figure 11.1a performs five vector loads to compute one result. We show the memory access footprint of one invocation of the vectorized kernel in Figure 11.1b.

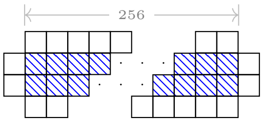
Contrast this with the tensorized version shown in Figure 11.2a where the stencil is tensorized in both the `i`-loop by 256 and the `j`-loop by 2 at once. We see two effects of tensorizing this kernel: First, since the native vector length of the target is 256, the tensor values of 2×256 elements are tiled into two vector registers. Second, our memory access chunking scheme implements tensor loads using fast contiguous vector loads. The memory loads of the tensorized code overlap, which is why `c0` and `e0` are equivalent to already loaded values. In effect, the tensorized kernel performs eight loads to compute two results on average where the vectorized kernel requires ten loads to achieve the same.

However, there is more to tensorization than register tiling: By considering an entire loop nest, we can emit fast contiguous memory accesses even if the memory instructions of a kernel are contiguous in different iteration variables. We demonstrate this for matrix transposition, that is the transformation `B(i, j) = A(j, i)`.

In the tensorized matrix transpose, shown in Figure 11.3, four contiguous loads and four contiguous stores are used along with shuffles to efficiently transpose 16 matrix elements in one go. This leads to a speedup of 6.11× over the loop vectorized version on a AVX512 platform in multi-threaded execution

```
for (j = 1; j < rows - 1; j += 2) {
  for (i = 1; i < cols - 1; i += 256) {
    a0 = load_v256(A(j-1,i  ));
    a1 = load_v256(A(j  ,i  ));
    b0 = load_v256(A(j  ,i-1));
    b1 = load_v256(A(j+1,i-1));
    c0 = a1; // reuse
    c1 = load_v256(A(j+1,i  ));
    d0 = load_v256(A(j  , i+1));
    d1 = load_v256(A(j+1, i+1));
    e0 = c1; // reuse
    e1 = load_v256(A(j+2, i ));

    b0 = .2 * (a0+b0+c0+d0+e0);
    b1 = .2 * (a1+b1+c1+d1+e1);
    store_v256(B(j  ,i), b0);
    store_v256(B(j+1,i), b1);
  }
}
```



(a) Jacobi tensorized in both j (width 2) and i (width 256) resulting in 20% less vector loads.

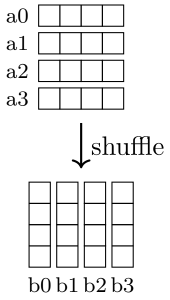
(b) In average 2 unique cells have to be loaded to compute one result.

Figure 11.2.: Tensorized 5-point Jacobi stencil.

```
for (j = 0; j < rows; j += 4) {
  for (i = 0; i < cols; i += 4) {
    // fetch contiguous chunks of A
    a0 = load_v4(A(j  , i);
    ..
    a3 = load_v4(A(j+3, i);

    // in-register transpose
    b0 = shuffle(a0[0], a1[0], a2[0], a3[0])
    ..
    b3 = shuffle(a0[3], a1[3], a2[3], a3[3])

    // store contiguous chunks to B
    store_v4(B(i, j),   b0)
    ..
    store_v4(B(i, j+3), b3)
  }
}
```



(a) Matrix transpose tensorized in both loops by 4 × 4. Loads are contiguous in i. Stores are contiguous in j.

(b) Efficient shuffle operations transpose a 4 × 4 tile of the matrix in one invocation.

Figure 11.3.: Tensorized matrix transposition kernel.

(Section 13.4). When vectorized in only one loop, there is only one contiguous access and the other would be a slow scatter or gather instruction.

11.1. Tensorization

Tensorization is the generalization of vectorization to multiple dimensions. The logical generalization of the one-dimensional number of threads is the tensor brush, as given by Definition 24.

Definition 24. *(Tensor Brush) A P-LLVM tensor program executes for a d-dimensional array of threads, whose sizes are given by the tensor brush:*

$$\mathcal{B} = (m_0 \times \cdots \times m_{d-1})$$

The value $m_i \in \mathbb{N}$ is the size of the brush in the direction of the i-th dimension.

In one-dimensional vectorization, the threads of the thread array are enumerated by their thread identifier $\mathtt{t} \in \mathcal{T} = \{ 0, .., W-1 \}$. In tensorization, we use *Coordinates* to refer to the multi-dimensional elements of a given brush. For example, $(1, 2)$ is a valid coordinate for the brush $\mathcal{B} = 4 \times 4$.

In loop-nest tensorization, each dimension of the brush corresponds to one loop level of the loop nest. The outer-most loop is always at dimension 0 increasing with each nested loop.

11.2. System Overview

Figure 11.4 gives an overview of the loop-nest tensorization pipeline. The main inputs to the system are the loop nest to tensorize, the tensor brush, which specifies the extent of the tensor in each loop and the brush projection, which describes how the coordinates in the tensor brush are mapped to SIMD registers.

The pipeline operates in four phases: First, the tensor shape analysis determines how each variable behaves in relation to the loops that surround it (Section 11.3). We also run LLVM's Scalar Evolution analysis to decompose address arithmetic into multi-dimensional array accesses with index vectors. We optionally permute index vectors to transform the data layout (Section 11.7). Second, we group memory accesses into strips of contiguous buffer elements (Section 11.4). Finally, the tensor shapes and memory access groups are used to generate SIMD code with optimized memory accesses (Section 11.6). During the SIMD code generation step, the brush projection determines how the coordinates in the tensor brush are mapped to the one-dimensional lanes of SIMD registers.

11.3. Tensor Shapes

The tensor shape analysis determines whether and how the instructions in the loop nest depend on the iteration variables of the loops that surround them. The analysis assigns to every instruction a tensor shape that captures the nature of this dependence.

The *tensor shape* describes per surrounding loop how the value of the instruction changes from one loop iteration to the next. This information is recorded separately for each dimension, similar to how the set of partial derivatives constitutes the complete derivative of a function. For each dimension of the loop nest separately, tensor shapes hold a vector shape defined as:

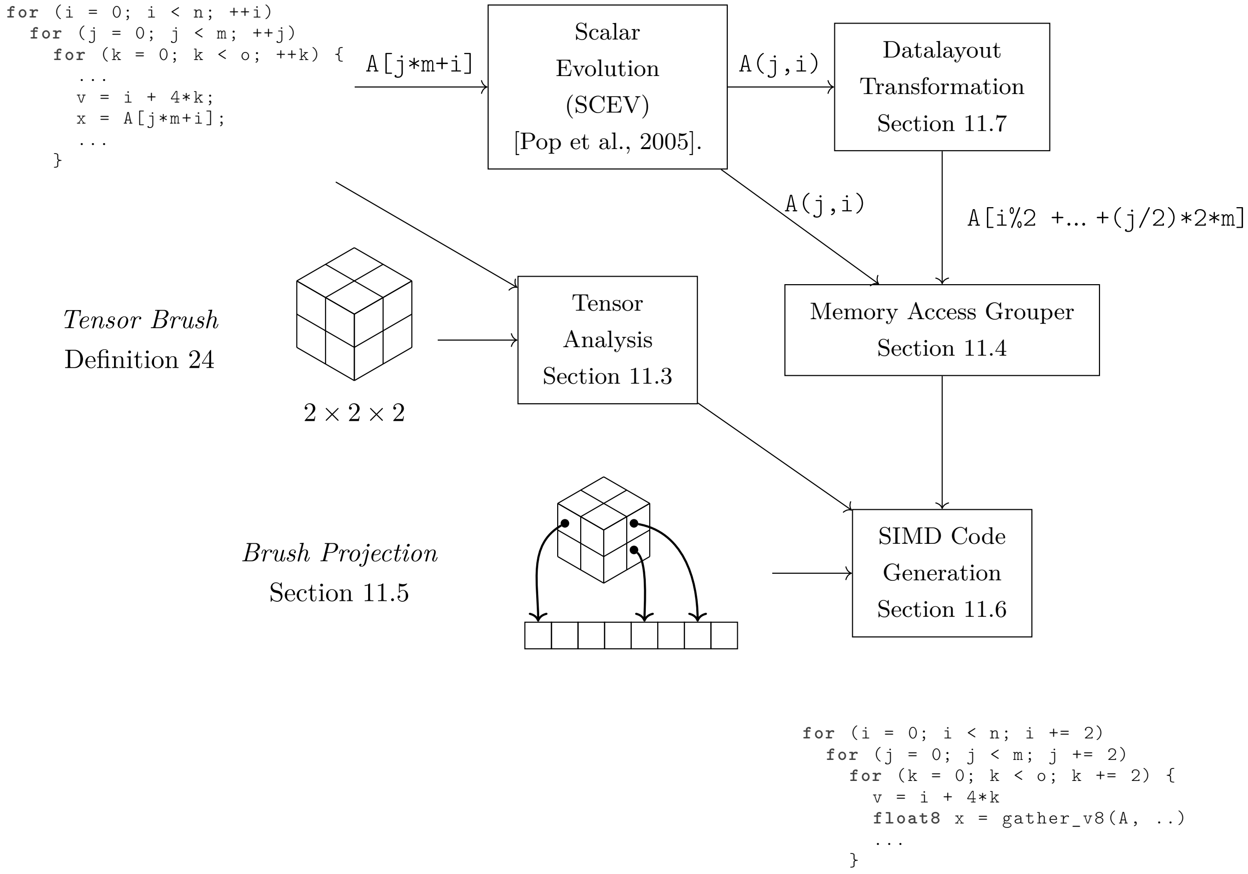
$$\mathcal{T}_{\mathtt{1D}} = \mathbb{Z} \cup \{ \top \}$$

The set of d-dimensional tensor shapes, $\mathcal{T}^d$, is then the composition of d one-dimensional vector shapes:

$$\mathcal{T}^d = (\mathcal{T}_{\mathtt{1D}})^d$$

We will use the notation $(s_0, s_1, s_2) \in \mathcal{T}^3$ to denote the elements of a $3D$ tensor shape. Each $s_i \in \mathcal{T}_{\mathtt{1D}}$ is the dimension shape of its dimension i.

If $s_i \in \mathcal{T}_{\mathtt{1D}}$ is an integer, then s_i is the constant loop increment for the annotated instruction for the loop at dimension i. In particular if $s_i = 0$ then the annotated instruction is invariant, that is *uniform*, in the loop. If $s_i \in \mathcal{T}_{\mathtt{1D}}$ is $\top$, then the instruction is *varying* in dimension i but the nature of the variation



```
for (i = 0; i < n; ++i)
  for (j = 0; j < m; ++j)
    for (k = 0; k < o; ++k) {
      ...
      v = i + 4*k;
      x = A[j*m+i];
      ...
    }
```

A[j*m+i]
Scalar Evolution (SCEV) [Pop et al., 2005].
A(j,i)
Datalayout Transformation Section 11.7
A(j,i)
A[i%2 +... +(j/2)*2*m]
Tensor Brush
Definition 24
2 × 2 × 2
Tensor Analysis Section 11.3
Memory Access Grouper Section 11.4
Brush Projection
Section 11.5
SIMD Code Generation Section 11.6

```
for (i = 0; i < n; i += 2)
  for (j = 0; j < m; j += 2)
    for (k = 0; k < o; k += 2) {
      v = i + 4*k
      float8 x = gather_v8(A, ..)
      ...
    }
```

Figure 11.4.: System overview of loop-nest tensorization.

```
for (i = 0; i < m; ++i) // dim 0
  for (j = 0; j < n; ++j) // dim 1
    for (k = 0; k < w; ++k) // dim 2
      ..S..
```

(a) A 3D loop nest with placeholder statement S.

..S..	Tensor shape $\mathcal{T}_S$
i	$(1, 0, 0)$
j	$(0, 1, 0)$
k	$(0, 0, 1)$
12 * i - 6 * j	$(12, -6, 0)$
2*A(i) + 10 * j	$(\top, 10, 0)$
B(i,j) + 5 * k	$(\top, \top, 5)$
C(i) * B(k)	$(\top, 0, \top)$
C(B(j))	$(0, \top, 0)$

(b) Upper section: initial tensor shapes, Lower section: tensor shapes inferred by the tensor shape analysis.

Figure 11.5.: 11.5a: A 3D loop nest with placeholder statement S. 11.5b: Examples for S and their resulting tensor shapes.

is not reflected in the tensor shape. For example, if i is used in an array load such as A[..] then the loaded value is always varying in i because the contents of the array are in general unknown.

Figure 11.5a shows a three-dimension loop nest and the tensor shapes of its induction variables are listed in the upper half of Figure 11.5b. The induction variables are invariant in all other dimensions than their own. For example, the induction variable `j` has a stride of 0, that is it is uniform, in all dimensions but dimension 1.

The tensor shape analysis propagates these initial shapes to all instructions in the loop nests. We show some examples for fill-ins for the placeholder `S` of Figure 11.5a in the lower half of Figure 11.5b.

If the instruction computes an affine combination of iteration variables, the tensor shape reflects these as strides. The tensor shape analysis assumes that there are no loop-carried dependencies between the instructions of the loop nest. Therefore, the tensor shapes of memory accesses only depends on the tensor shape of the address computation. The relation between values loaded from different addresses is, however, *varying*.

11.4. Memory Access Grouping

If memory accesses are naively widened, there is often an overlap in the accessed elements among different instructions. Memory access grouping is mapping out this overlap with the end of generating more efficient vector memory accesses in the code generation phase.

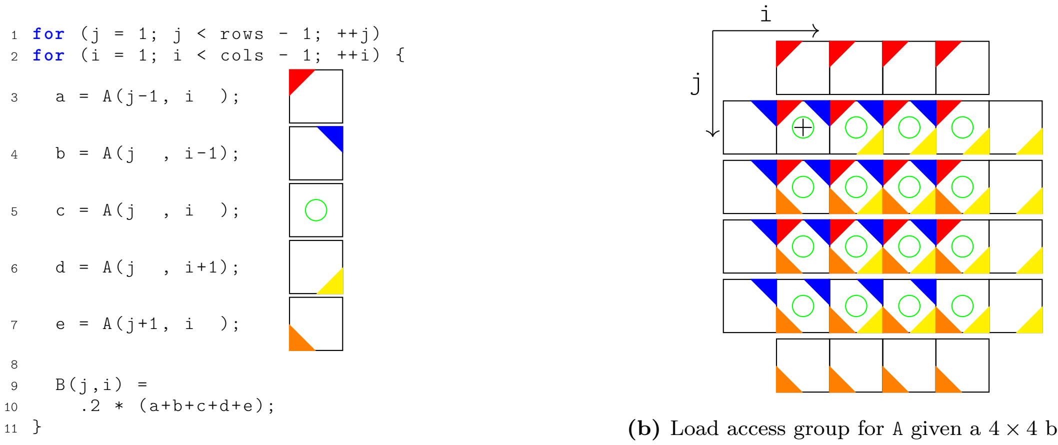
An access group is defined by the equivalence relation $\overset{\mathtt{M}}{\sim}$ on memory instructions plus coordinate pairs. If $(\mathtt{L}, c) \overset{\mathtt{M}}{\sim} (\mathtt{L}', c')$ then the value loaded by `L` at tensor coordinate c will be the same as the one loaded by `L'` at coordinate c'. For example, it holds that $\mathtt{A(i-1,j)}, (2,0) \overset{\mathtt{M}}{\sim} \mathtt{A(i,j)}, (1,0)$.

Memory access grouping is concerned with constructing these access groups for all memory accesses in the loop nest. Since the tensor brush has finite extent and there are only finitely many memory instructions in the kernel, we can explicitly construct each access group by enumeration. The implementation builds on LLVM's *Scalar Evolution* analysis to compute the offsets between pointers [Pop et al., 2005].

Figure 11.6a shows the memory access grouping of a 5-points Jacobi stencil. Assume that the loop nest will be vectorized for a 4×4 tensor brush (Definition 24). Figure 11.6b shows the resulting access group for the loads in Line 3 to Line 7. We use a color/symbol coding scheme in the figure to visualize each load instruction. To the right, in Figure 11.6b, we show the accessed elements of `A`. Each cell in Figure 11.6b is an array subscript to `A`. The symbols in the cells indicate which positions of the array are accessed by which load instructions.

As an example consider the cell marked '+', which refers to the array element `A(j,i)`. The cell contains the instruction symbols of Line 3, Line 4 and Line 5. We can read off the access group that Coordinate $(1,0)$ of the load in Line 3 refers to the same pointer as Coordinate $(0,1)$ of the load in Line 4.

For our prototype, we assume that there are no conflicts between loads and stores, e.g. that the elements fetched by a load would become invalid due to an overwriting store. This is a typical pattern for stencil codes, which fetch elements from an input array and store the accumulated sum into an output array.



```
1  for (j = 1; j < rows - 1; ++j)
2  for (i = 1; i < cols - 1; ++i) {
3    a = A(j-1, i  );
4    b = A(j  , i-1);
5    c = A(j  , i  );
6    d = A(j  , i+1);
7    e = A(j+1, i  );
8
9    B(j,i) =
10     .2 * (a+b+c+d+e);
11 }
```

(a) 5-point Jacobi kernel

(b) Load access group for `A` given a 4×4 brush. Elements in `i` direction are contiguous, the distance in `j` direction is m, indicated by a gap.

Figure 11.6.

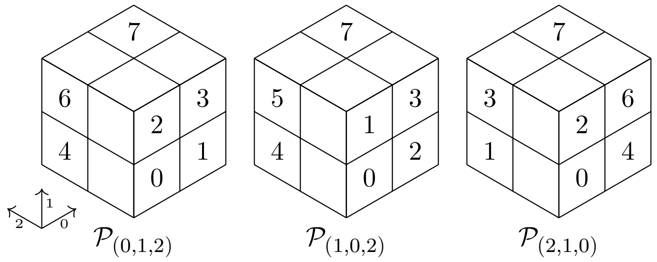
11.5. Projecting Coordinates to SIMD Lanes

SIMD ISAs do not actually have tensor registers but one-dimensional vector registers. The tensor brush only specifies the size of the tensor in its dimensions. There is ambiguity in how the coordinates in a tensor are mapped to the lanes of SIMD registers. We resolve this ambiguity with the brush projection.

A *brush projection* ($\mathcal{P}_{\mathcal{B}}$) defines a unique mapping from the multi-dimensional coordinate space of the vector brush $\mathcal{B}$ to vector lanes. The brush projection is defined by the projection vector, a vector of dimension numbers $(i_0, \ldots i_{d-1})$. The projection vector specifies a cardinality of the dimensions. The brush coordinates are then enumerated according to this order similar to numbers in positional notation. We define the projection from coordinates $(c_0, .., c_{d-1})$ to vector lanes, given a brush $\mathcal{B} = (m_0 \times .. \times m_{d-1})$ and a projection vector $(i_0, .., i_{d-1})$ as follows:

$$\mathcal{P}_{(i,i_1,..,i_{d'})}(c_0, .., c_{d-1}) = \left\{ \begin{array}{cc} c_i + m_i \mathcal{P}_{(i_1,..,i_{d'})}(c_0, .., c_{d-1}) & \text{if } d' > 1 \\ c_i & \text{otw} \end{array} \right\}$$

We show three brush projections of a $2 \times 2 \times 2$ brush in Figure 11.7.



(a)

$\mathcal{P}_{(0,1,2)} =$	(0,0,0)	(1,0,0)	(0,1,0)	(1,1,0)	(0,0,1)	(1,0,1)	(0,1,1)	(1,1,1)
$\mathcal{P}_{(1,0,2)} =$	(0,0,0)	(0,1,0)	(1,0,0)	(1,1,0)	(0,0,1)	(0,1,1)	(1,0,1)	(1,1,1)
$\mathcal{P}_{(2,1,0)} =$	(0,0,0)	(0,0,1)	(0,1,0)	(0,1,1)	(1,0,0)	(1,0,1)	(1,1,0)	(1,1,1)

(b)

Figure 11.7.: Vector lane mapping according to brush projections. 11.7a: Vector lane numbers drawn onto tensor coordinates. 11.7b: Respective coordinates at vector lanes.

11.6. SIMD Code Generation

This section describes the generation of SIMD instructions for a tensorized loop nest starting from scalar code and the analysis data that has been gathered up to this stage. The analysis data are tensor shapes (Section 11.3) and memory access groups (Section 11.4). The loop trip counts are trivially transformed by scaling them by the brush sizes.

Section 11.6.1 describes the algorithm for widening code with tensor shapes to SIMD instructions and Section 11.6.2 covers how we efficiently vectorize tensor memory accesses.

11.6.1. Widening with Tensor Shapes

We widen tensorized loop nests into SIMD code with the following procedure. The algorithm visits every instruction in the loop nest in reverse post-order and emits SIMD code according to the tensor shapes of the instruction. We deal with memory access separately (Section 11.6.2). The generic scheme distinguishes two cases for the vector shapes: all-strided and one-varying. We assume that all instructions except memory accesses are free of side effects.

All Strided. The first case applies when the tensor shape is strided (or uniform) in all dimensions with brush size greater than 1. In this case, we emit a scalar instruction that computes the result of the instruction at the zero coordinate $(0, .., 0)$. Since the operation is side-effect free, there is no need to replicate it for all coordinates, we only have to provide the computed result for all brush coordinates. Finally, we can compute the results at all other coordinates whenever needed simply by adding an appropriate multiple of the strides.

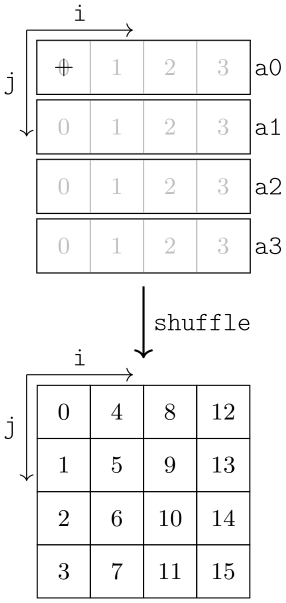
One Varying. If the tensor shape of an operation is $\top$ in at least one dimension, the scalar data type will be widened to a vector data type. The length of the vector is the product of all dimension sizes of the brush $\mathcal{B}$. For example, if the brush is $4 \times 2 \times 4$ and the data type of the operations is `double`, the widened vector instruction will operate on the data type `<32 x double>`. If any of the operands of the operation fell in the *all-strided* case they will now be instantiated into the lanes of a full vector according to the brush projection, $\mathcal{P}$.

Vector Register Tiling. We leverage the legalization phase of LLVM to effectively implement vector register tiling. During legalization, the LLVM compiler splits SIMD operations and variables to make them fit into the SIMD registers of the target. Tiling by leglization is triggered if the number of coordinates in the brush is greater than the vector register length. For example, if $\mathcal{B} = 2 \times 8$ and the element type is `double`, our code generation phase will initially emit a 16-element vector instruction in LLVM IR. However, the SIMD register length for `double` of the AVX512 SIMD ISA 8 and so LLVM will replace the instruction by two vector instructions with 8 elements.

11.6.2. Memory Access Chunking

Memory chunking is the process of partitioning the accessed elements into contiguous parts, called *chunks*. Each chunk is then loaded or stored with a fast contiguous memory instruction.

On the top right of Figure 11.8, we show a single memory access in a loop nest of depth two. Consider that this loop nest is tensorized with a brush of 4×4. Using the generic one-varying strategy, the load would be vectorized with a slow gather instruction. However, scatter and gather instruction are the least



```
// scalar loop nest
for (i=0; i<k; ++i) // dim 0
for (j=0; j<m; ++j) { // dim 1
  x = A(j, i);
  ...
}
```

```
// tensorized loop nest
for (i=0; i<k; i += 4) // dim 0
for (j=0; j<m; j += 4){// dim 1
// chunked load
  a0 = vload_4(A(j,   i));
  a1 = vload_4(A(j+1, i));
  a2 = vload_4(A(j+2, i));
  a3 = vload_4(A(j+3, i));

// shuffling
  x = shufflevector [
  a0[0], a1[0], a2[0], a3[0],
  a0[1], a1[1], a2[1], a3[1],
  a0[2], a1[2], a2[2], a3[2],
  a0[3], a1[3], a2[3], a3[3],
  ];
  ...
}
```

Figure 11.8.: Chunking a tensor load into contiguous memory accesses for $\mathcal{B} = 4 \times 4$. **Left:** The load `A(i, j)` is chunked into contiguous loads `a0` to `a1`. Below, the loaded chunks are shuffled according to the brush projection $\mathcal{P}_{\mathcal{B}} = (1, 0)$. Numbers in the cells refer to the lane number. **Right:** The scalar loop nest on top. Below, the tensorized version with chunked loads and shuffles.

effective means of accessing memory on AVX512. It is advisable to use contiguous memory accesses whenever possible.

We build memory access groups as presented in Section 11.4 and partition the groups into strips of contiguous memory accesses. This results in the contiguous loads `a0` to `a3` shown below. This is an AVX2 example (256*bit* SIMD registers) and thus each consecutive vector load can transfer four contiguous element from the `double` array.

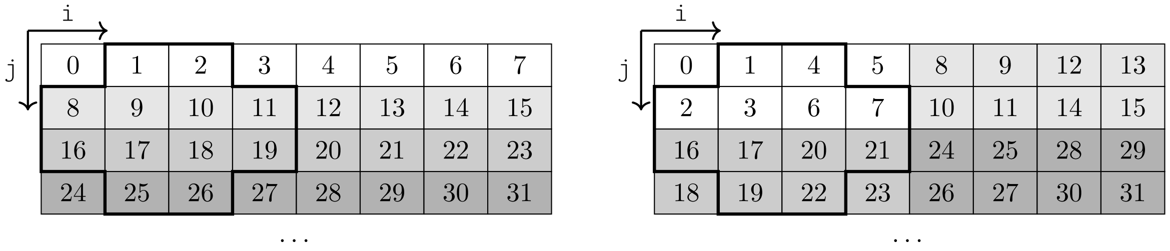
Finally, the lanes need to be shuffled into the vector according to the brush projection. We do the inverse, shuffling followed by consecutive stores, to vectorize the tensor store.

11.7. Data Layout Transformation

Stencil codes benefit from tensorization by exposing opportunities for reuse of already loaded values, as discussed in Section 11.4. This section describes a technique to further optimize SIMD loads by reorganizing the data in the image buffers through a data layout transformation. Consider Figure 11.9, that shows two different buffer layouts.

Standard Layout. Figure 11.9a shows the standard layout of a two dimensional image buffer. Rows are layed out contiguously in memory along the dimension of `i`. We overlap the memory layout with the load footprint of a 5-point jacobi stencil tensorized with a 2×2 brush. As the diagram shows, four SIMD loads are needed to retrieve the data necessary to compute all four result elements.

2×2 Layout. Figure 11.9b shows the layout of the input data after a data layout transformation. Here, the image is subdivided into blocks of two by two elements, which are layed out contiguously



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

(a) Standard contiguous memory layout (1 × 1). Four SIMD loads required.

0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31

(b) Transformed data layout (2 × 2). Two SIMD loads required.

Figure 11.9.: Comparison of memory layouts of a 2D image buffer `A(j,i)` of `double` elements. Element ranges that are loaded by a single contiguous SIMD load (AVX512) are color coded. The overlayed outline indicates the load footprint of jacobi 5-point stencil tensorized for a 2×2 brush.

in memory. The ordering of blocks is again contiguous in the dimension of `i`. We again overlap the diagram with the footprint of the 5-point jacobi stencil. Thanks to the changed layout, the footprint only overlaps the elements of two SIMD loads.

Implementation. The memory access grouping phase (Section 11.4) builds on LLVM's *Scalar Evolution* analysis [Pop et al., 2005] (SCEV). With the SCEV analysis an array access `A[m*j+i]` can be decomposed into an access at index (j, i) into buffer `A`. Transforming the data layout then means constructing a new SCEV expression from that decomposition. Since the transformation modifies the SCEV representation and all other parts of the system that touch memory accesses leverage SCEV, this data layout transformation is completely transparent.

Specifically, consider switching from the standard layout shown in Figure 11.9a to the 2×2 layout shown in Figure 11.9b for a given buffer `A`. Every memory access to `A` needs to decompose to the base pointer `A` and some index vector (j, i) with m being the length of a row. The transformed SCEV is then computed as: `A[i%2 + (i/2)*4 + (j%2)*2 + (j/2)*2*m]`.

11.8. Related Work

Stratton et al. [2010] describe variance vectors attached to instructions to classify the variance of variables with respect to the work item dimensions of CUDA kernels. Each dimension that the instruction varies in is an element of the variance vector, meaning that the approach relates to the tensor shapes presented in Section 11.3 as a form of basic multi-dimensional divergence lattice. The variance vector is not used for actual multi-dimensional code generation. The tensor shape analysis is a multi-dimensional generalization of Chapter 6.

There is limited support for multi-dimensional vectorization in the ISPC programming language [Pharr and Mark, 2012] through the `foreach_tiled` statement. However, lacking a multi-dimensional divergence analysis, ISPC will use scatter/gather to vectorize every memory access that is not fully uniform in that mode.

Plagne and Bojnourdi [2017] present a multi-dimensional vector code generation library based on C++ template programing. Library-based techniques require the programs to be specially written whereas our approach starts from regular LLVM IR.

Since most modern systems have single dimensional memory, even contiguous accesses in higher dimensions lead to strided memory accesses that require gather and scatter instructions. This was addressed in Vector Folding [Yount, 2015] by performing data layout transformations as a preprocessing step before multi-dimensional vectorization. Data layout transformations have also been used Henretty et al. [2011] to reduce shuffle operations induced by unaligned memory accesses. In our approach, we use shuffles with contiguous loads to generate operand vectors [Caballero et al., 2015; Eichenberger et al., 2004].

The polyhedral model [Feautrier, 1992b] is only applicable in loop nest tenorization if all branch conditions are piecewise affine functions in the iteration vector. However, all transformations and analyses of RV are generally applicable in the multi-dimensional context of TensorRV.

Chapter 12.

Related Work

In this chapter, we compare the RV vectorization system as a whole to other systems that have been proposed in the literature.

12.1. SIMD Programming Languages and Compilers

SIMD programming languages offer scalar data types and guarantee that the language compiler emits SIMD code such that each lane realizes one instance of the code. This is the programming model of P-LLVM but as a structured programming language.

Systems of this domain are the Intel SPMD Program Compiler (ISPC) [Pharr and Mark, 2012] and the Sierra language compiler [Leißa et al., 2014]. The ISPC and Sierra compilers operate exclusively on the AST representation of the program. These systems are monolithic in the sense that they emit SIMD code directly from the AST without any intermediate representation. Transformation phases exist only in so far as the AST is decorated with analysis results. This is problematic as transformations are restrained by the structural limits of the AST, e.g. it is only possible to insert BOSCC (skip-all-false branches) over nested AST nodes.

Further, ISPC and Sierra are domain-specific languages and programs in general purpose languages such as C/C++ or Fortran have to be ported to leverage them. This is complicated by the fact that even though Sierra and ISPC might have a C-like syntax the language semantics differs in subtle ways, i.e. in case of automatic type conversion [Pohl et al., 2016]. An alternative approach is to re-structure a program given in compiler IR to use the existing vectorization techniques of ISPC and Sierra. We discuss in Section 12.7 why re-structuring is not ideal for vectorizing general CFGs.

12.2. Explicit SIMD Programming

Compilers expose target-specific SIMD types and instructions as builtin types and functions. Programming with SIMD intrinsics is tedious and non-portable across SIMD architectures. Several SIMD programming libraries have been developed that offer a portable interface abstracting from specific target intrinsics [Kretz, 2015; Estérie et al., 2012; Georgiev and Slusallek, 2008]. Such a programing model has recently been standardized for the C++ language with the parallelism v2 TS [ISO 19570:2018].

Extensions have been proposed for Fortran and C/C++ to target SIMD CPUs through the abstraction of arrays. This was first done for Fortran [Albert et al., 1988; Hendrickson, 1979; Guzzi et al., 1990] and re-emerged for C/C++ with Intel Array Building Blocks [Newburn et al., 2011]. Array extensions provide array types with SIMD-like operations, including structured forms of element shuffling and explicitly masked SIMD operations.

However, explicit SIMD programming entails explicit handling of divergent control flow and manual inference of uniform and varying values. In comparison, the RV programming model accepts any scalar code and performs divergence analysis and transformations automatically.

One telling example of this is the Embree raytracer [Wald et al., 2014]. The raytracing codes in Embree use explicit SIMD programming with compiler intrinsics to furnish specialized code paths for every recent x86 SIMD ISA (AVX2, AVX512). This is in contrast to the raytracer generator Rodent [Pérard-Gayot et al., 2019]. Rodent leverages, among other techniques [Leißa et al., 2018], the RV vectorization system for implicit SIMD programming. The raytracing parts specifically rely on whole-function vectorization through the RV vectorization system to generate SIMD code. Pérard-Gayot et al. [2019] show that the generated raytracing codes perform on par with the handwritten intrinsic codes of the Embree system while being portable to non-X86 platforms.

12.3. Vectorizers for Superword-Level Parallelism

Vectorizers for Super-word level Parallelism (SLP) originally [Larsen and Amarasinghe, 2000] group independent scalar operations into SIMD operations. Several enhancements have been proposed, for SLP in the context of control flow [Shin et al., 2005], improved cost models [Mendis and Amarasinghe, 2018; Porpodas and Jones, 2015] and by relaxing constraints on legal instruction groupings [Porpodas et al., 2015; Mendis et al., 2019].

All of these techniques intrinsically build on the notion of independence of instructions under sequential execution. As such, SLP vectorization is also applicable in the data-parallel setting by grouping independent instructions within a thread. It has been noted, however, that outer-loop vectorization can be framed as unroll-and-jam of a loop nest followed by SLP vectorization [Nuzman and Zaks, 2008]. Yet, being limited to scalar instruction instances, standard SLP vectorizers do not implement data-parallel vectorization.

Further, grouping instruction instances across loop iterations has been explored in the context of loop vectorization: Zhou and Xue [2016] present an approach that mixes SLP vectorization with loop vectorization techniques. Nuzman et al. [2006] and Anderson et al. [2016] present techniques to generate efficient SIMD code for interleaved scalar memory accesses.

Being defined in the setting of sequential control, none of these approach consider data-parallelism or horizontal operations. However, SLP and inter-leaving techniques can be used as optimizations in widening stage of a data-parallel vectorizer. Consider, e.g. the data re-use optimizations in TensorRV (Chapter 11).

12.4. Whole-Function Vectorizer

Karrenberg and Hack [2011, 2012]; Karrenberg et al. [2013]; Karrenberg [2015] pioneered whole-function vectorization for SSA-based CFGs. The whole-function vectorizer is described as a pass-based vectorization pipeline similar to the RV system. However, WFV does not have one data-parallel intermediate representation but maintains aspects of semantics in separate data structures that are not considered part of the program. This includes, for example, a data structure to describe the masking code of basic blocks. RV is designed the other way round by putting the program representation, P-LLVM, at the core of the vectorizer. P-LLVM too is implemented with overlays and data structures on top of LLVM IR but there is one crucial difference: The data structures serve to implement P-LLVM, which gives each program a defined semantics. The RV vectorizer is thus a pipeline of combinable passes that each transform P-LLVM programs.

12.5. Loop Vectorizers

Loop nests of scalar operations have long been the target of automatic vectorizers.

Early automatic loop vectorizers used source-to-source translation [Allen and Kennedy, 1987]. Ngo [1995] defines the `DOVEC` loop statement, which models lock step execution in structured loop nests. However, the statement is only used for the purposes of correct dependence modelling not code transformation.

Loop vectorizers are typically restricted to uniform loop nests [Nuzman et al., 2006; Nuzman and Zaks, 2008], otherwise resorting to if-conversion. Even if a loop is annotated as parallel, for example by means of OpenPM pragmas, this may not mean more than that the vectorizer can assume that there are no loop-carried dependences. Horizontal operators are, due to the lack of a proper data-parallel semantics for loops, unavailable in standard loop vectorizers.

Masten et al. [2018] propose a technique to implement whole-function vectorization with an outer-loop vectorizer. In their approach, the scalar function body is wrapped in a loop, which is then inserted into the vector function declaration. While this offers whole-function vectorization with an outer-loop vectorizer, the approach does not solve the problem of vectorization.

Program Dependence Graph. Various intermediate representations have been proposed to expose the latent parallelism in programs. The re-occurring theme is to dissolve the strict execution order imposed by imperative programming languages, or the Control-Flow Graph. Among these representations are the Program Dependence Graph (PDG) [Ferrante et al., 1987] and the (Regionalized) Value State Dependence Graph ((R)VSDG) [Johnson and Mycroft, 2003] and several variations thereof, such as the Parallel Program Graph (PPG) [Sarkar, 1992].

The Program Dependence Graph (PDG) [Ferrante et al., 1987] has been used as a representation for vectorizing compilers [Baxter and III, 1989]. This was foremost motivated by the need to identify parallel loops. Simons et al. [1990] noted that transforming a PDG into a sequential CFG is a non-trivial task. Generating a CFG from the PDG is, however, necessary since the branch instructions of CPUs implement unstructured, sequential control-flow. This problem does not exist in the RV vectorization system since P-LLVM is CFG based.

Polyhedral Model. The polyhedral model [Feautrier, 1991, 1992a,b] represents loop nests as affine polyhedra. This is possible if all branch conditions and loop bounds are piecewise affine functions of the iteration variables. Several vectorization systems have been developed that leverage the polyhedral model to vectorize loop nests [Kong et al., 2013; Trifunovic et al., 2009] and OpenCL kernels [Moll et al., 2016]. These techniques deliver good results if the loop nest is representable in the polyhedral domain. Otherwise, vectorization fails since polyhedral techniques are not applicable to general codes.

12.6. Specialized Code Generators

The importance of efficient codes for certain computational problems warrants the design of specialized code generators that also leverage SIMD instructions. Among these are works on tree traversal codes [Jo et al., 2013], dense and sparse matrix multiplication [Heinecke et al., 2016; Spampinato et al., 2018], sorting networks [Hou et al., 2015] and FFTW [McFarlin et al., 2011] to name a few. These approaches excel in their target domain but are not applicable to the vectorization of general codes.

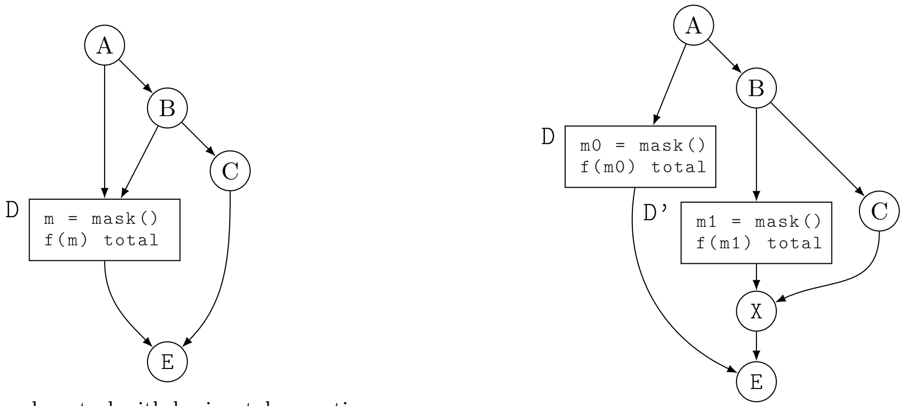
Several frameworks have been proposed with data-parallel code representations with the end of transforming the program in the compiler. The Lift IR Steuwer et al. [2017] is a functional compiler IR that translates to OpenCL code in the Lift compiler backend. It is therefore on the OpenCL driver to perform the actual vectorization. The Halide IR [Ragan-Kelley et al., 2013] is a image processing DSL and compiler. Both frameworks can express parallel loops and, in case of Halide, loop vectorization. However, Halides loop vectorizer does not consider control flow.

12.7. Restructuring Vectorizers

Techniques for the vectorization of structure code are available, e.g. in the ISPC compiler [Pharr and Mark, 2012]. Many of the analyses and transformations required in a vectorizer become simpler when only considering structured codes [Leißa, 2017; Reiche, 2018].

The common notion of structured control flow [Sharir, 1980] is that the code decomposes into single-entry single-exit (SESE) subgraphs. Each SESE region has the control-flow of a canonical, structured control-flow element (IFTHEN, IFTHENELSE, BLOCK). The children of these structured elements are again SESE subgraphs.

Transformations that are simpler under the assumption of structured control are the divergence analysis and control-divergence analysis, partial if-conversion schemes and techniques similar to the BOSCC gadget proposed in Section 9.2. Hence, it may be tempting to restructure the CFG to employ the existing techniques for structured code. Besides the fact that restructuring distorts the original program flow, the question remains whether this would produce adequate results.



(a) Unstructured control with horizontal operation.

(b) Semantic change after splitting the node D.

Figure 12.1.: Re-structuring by node splitting violates the semantics of horizontal operators in P-LLVM.

We consider two restructuring techniques: The first is the classic node splitting technique [Hecht, 1977]. The second is tail predication. Tail predication was recently proposed by Reissmann et al. [2016] to restructure Control-Flow Graphs without extra blocks [Reissmann et al., 2016, Fig. 3]. The example CFGs for our discussion are shown in Figure 12.1a and Figure 12.2a.

Node Splitting. Consider the unstructured P-LLVM program in Figure 12.1a. The DAG is unstructured because the block D is an immediate successor of B but B does not it nor does D post-dominate B. Node splitting creates one copy of the block for each incoming edge. We split the block D, resulting in the CFG of Figure 12.1b. We also insert a new empty basic block X to form a proper SESE region from B to X. The DAG in Figure 12.1b is structured. However, the restructured program has a different behavior. The block D contains a mask query and the result of that query is passed into a total function call. When the block D is split into D and D', the mask query and the function call are also duplicated. Clearly, if the branch in A is divergent then the two programs have a different semantics. In the unstructured

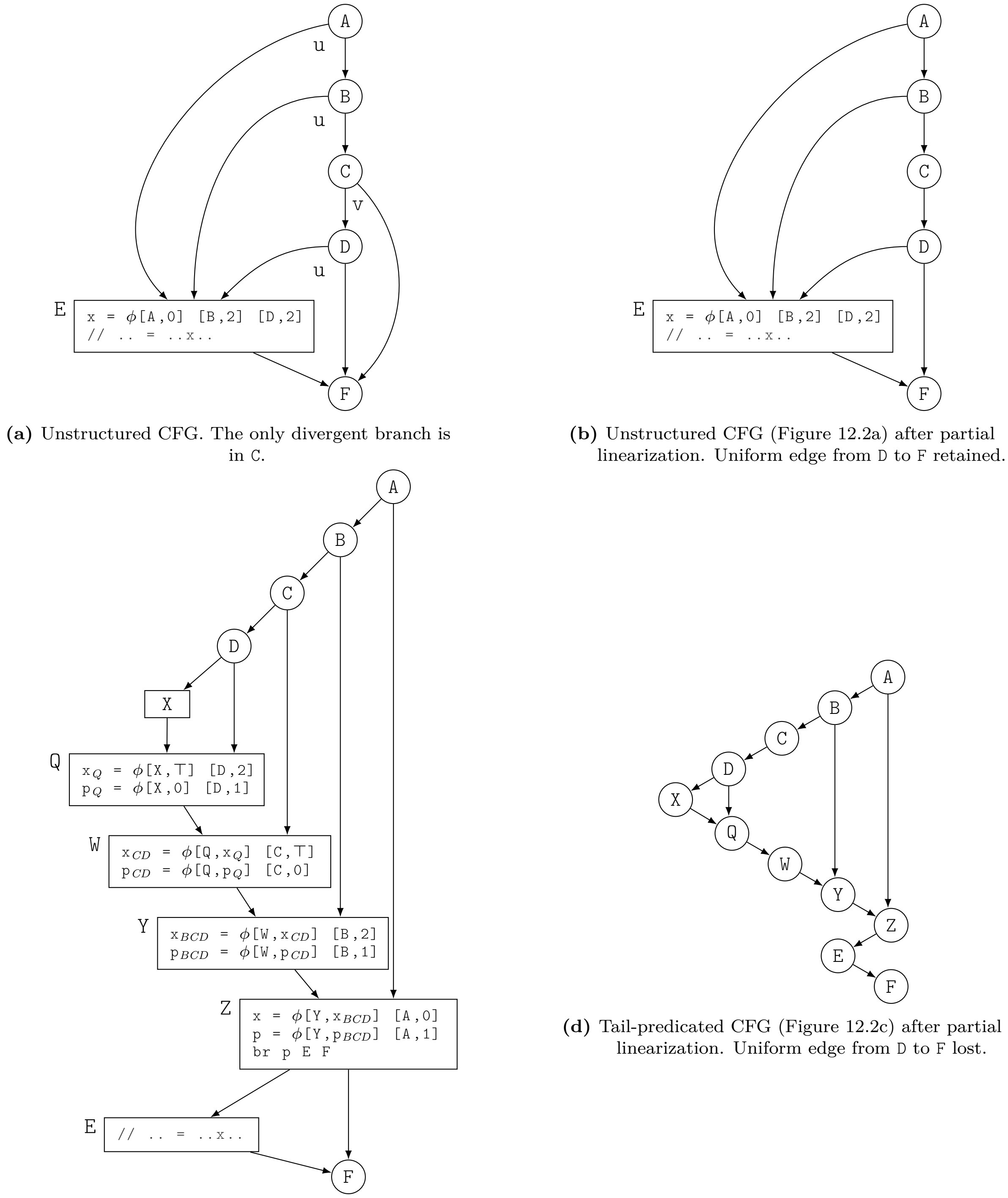
CFG, since all valid schedules are greedy schedules, it is guaranteed that the function call in D executes only once. After splitting, the function call is split in two and the f function may be called twice for different active threads and even with different arguments than before. In short, node splitting is not generally applicable in the context of data-parallel programs.

Tail Predication. Tail predication is an alternative technique to restructure programs. Consider the unstructured DAG shown in Figure 12.2a. This CFG is restructured with tail predication in Figure 12.2c. In tail predication, blocks that violate structuredness are put on a new branch controlled with a new boolean variable. Even though the existing blocks are not split, the transformation still comes at the price of additional blocks and instructions: Blocks are added only for the sole purpose of establishing control structure, instead of one ϕ node, there are now *eight*.

Setting aside code size considerations, tail predication affects the result of partial linearization. For the sake of the example, assume that all branches in the CFG are uniform except for the divergent branch in C. Figure 12.2b shows the original CFG of Figure 12.2a after partial linearization and Figure 12.2d the result of partially linearizing the restructured CFG in Figure 12.2c. In Figure 12.2b, partially linearization preserves the control-flow edge of D that jumps to F, skipping E. In Figure 12.2d, the block E always executes.

Practical Considerations. Reissmann et al. [2016] analyzed the 240 data-parallel kernels of Rodinia [Che et al., 2009] for their control-flow structuredness. They did not find any irreducible loop. However, they did find 11 benchmarks with unstructured, reducible control flow. This is consistent with an earlier survey on structure in GPU kernels [Wu et al., 2012], which found unstructuredness but no irreduciblity in Parboil [Stratton et al., 2012], the Optix render [Parker et al., 2010] and the CUDA benchmark suite. This empirical evidence backs the control-flow assumptions in this thesis: in real codes, control-flow is often unstructured, however, irreducible loops are rare.

Conclusion. To summarize, re-structuring seems tempting at first since it simplifies the control-divergence analysis and partial if-conversion. However, there is a hidden cost: the process of re-structuring itself already degrades the CFG leading to inferior overall outcomes in detected uniform variables (Figure 12.2) and specifically retained control flow (Figure 12.2d). Earlier surveys support the control-flow assumptions that we make in this thesis: control flow is often unstructured but irreducible loops are rare.



(a) Unstructured CFG. The only divergent branch is in C.

(b) Unstructured CFG (Figure 12.2a) after partial linearization. Uniform edge from D to F retained.

(c) CFG of Figure 12.2a after tail predication, which puts E on a new conditional branch.

(d) Tail-predicated CFG (Figure 12.2c) after partial linearization. Uniform edge from D to F lost.

Figure 12.2.: Tail-predication restructures CFGs at a potential loss of uniform control flow.

Chapter 13.

Evaluation

We evaluate the RV vectorization system with respect to the following questions.

- How does the RV vectorization system compare to other data-parallel vectorizers? We evaluate RV on a range of tree traversal codes and compare it against the ISPC SIMD code generator and a scalar baseline (Section 13.1).
- How relevant is RV for actual application codes? We present two case studies: the `nab_s` benchmark of SPEC2017 (Section 13.2) and XSBench (Section 13.3).
- What are the performance benefits of tensorization and how performance-sensitive is the technique to the brush? We evaluate the TensorRV systems on five different stencil codes and matrix transpose under various brushes and two different daya layouts (Section 13.4).

Evaluation Systems. We run the experiments on four different platforms, comprising four different microarchitectures and three different SIMD ISAs.

- **ARM-A72** ARM Cortex-A72 cores of a Rockchip RK3399 system (NEON, 128 bit SIMD registers, 2 cores).
- **ARM-A53** ARM Cortex-A53 cores of a Rockchip RK3399 system (NEON, 128 bit SIMD registers, 4 cores).
- **Ryzen-2** AMD Ryzen 5 2600X Six-Core Processor (AVX2, 256 bit SIMD registers, 6 cores, 12 threads). Turbo boost disabled.
- **Skylake-X** Intel Core i9-7900X CPU @ 3.30GHz (AVX512, 512 bit SIMD registers, 10 cores). Hyperthreading and turbo boost disabled.

Compilers. We evaluate the system with the following compilers.

- **GCC (9.1.0)** GNU C Compiler.
- **ICC (19.0.4.243)** Intel C Compiler. This compiler is only available on the Intel Skylake-X machine.
- **Clang (llvm/trunk 369781)** Default Clang O3 pipeline.
- **RV** Clang pipeline with RV but without LLVM LoopVectorizer or LLVM SLPVectorizer. The RV passes are inserted at the `EP_VectorizerStart` extension point of the LLVM pass pipeline.
- **ISPC (1.12.1dev)** Intel SPMD Program Compiler. The compiler is built with the same LLVM version as Clang and RV. The ISPC compiler only supports specific combinations of SIMD

ISAs, vector widths and data sizes (`--target=<t>` switch). These are: `avx2-i32x8`, `avx2-i64x4`, `avx512skx-i32x16` and `neon-i32x4`.

- **AOCC (2.0.0-Build191)** AMD Optimizing C/C++ Compiler. This compiler is only available on the Ryzen-2 machine.

All compilers were configured for 1-ULP error bound on the vector math functions. RV uses the vector math functions from the SLEEF 3.2 vector math library [Shibata et al., 2019].

	Branches		**Loops**		**Lp. Exits**		**Load Masks**			**Store Masks**			**Allocas**	
	uni	*div*	*uni*	*div*	*uni*	*div*	*none*	*uni*	*var*	*none*	*uni*	*var*	*uni*	*var*
mpc	16	4	5				5	23		6	12		5	
mpc9	20	10	7	2	2	4	3	19	14	4	7	2	20	
kd	6	12	2				11	10		2	13		3	
kd9	10	18	4	2	2	4	9	26	32	5	21	20	19	
pc	4	12	2				11	10		2	10		3	
pc9	8	18	4	2	2	4	9	26	32	5	20	20	18	
vp	3	1	1				13		1	2	3		1	
vp9	3	1	3				37		1	23	3		5	
bh	3		2				16	4		5	2		1	
bintree	5	3		1	1	1	1		4	2		2	1	
kmeans	6	12	2				11	10		2	13		3	
kmeans9	10	18	4	2	2	4	9	26	32	5	21	20	19	
xsbench	2	2		1		1	1		1	1				
nab/rv/1	2	7	1				5	1	6			4		
nab/boscc/1	5	7	1				5	1	6			4		
nab/rv/2	2	2	1				21	3	15	5	1	4		
nab/boscc/2	2	2	1				21	3	15	5	1	4		
nab/rv/3	1	7	1				26	4	17	5	1	4		
nab/boscc/3	4	7	1				26	4	17	5	1	4		

Figure 13.1.: Vectorization statistics for the benchmarks. *Lp. Exits* is referring to the exits of divergent loops. *Load and Store Masks* are categorized as having the activation 1 (*none*), a uniform activation (*uni*) or a varying activation (*var*).

Figure 13.1 shows code statistics for the RV-vectorized versions of the benchmarks.

13.1. Tree Traversal Codes

We evaluate *RV* on a set of tree traversal codes. The codes are stack based and feature mixed uniform and divergent branches, which exercise some components of RV in particular. The Divergence Analysis has to prove the uniformity of the traversal stacks. This happens in interplay with the Control Divergence Analysis (Chapter 7) and Alloca SSA (Section 6.6). Some of the kernels contain divergent loops that are handled by the divergent loop transform (Section 9.1). Finally, partial linearization (Chapter 8) is required to transform the divergent branches, retaining uniform control flow.

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* [Kulkarni et al., 2009] and *Treelogy* [Hegde et al., 2016] 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 [Aila and Laine, 2009], a well-known technique to increase SIMD utilization for such codes. The following list describes the benchmarks and their input sets in further detail:

- **bh** 3D n-body simulation using the Modified Barnes-Hut algorithm by Barnes [1990]. The underlying acceleration structure is an OctTree. *random:* 1000,000 random bodies. *plummer:* 100,000 bodies from a plummer model.
- **kd** Nearest Neighbor search (1-NN) with a kd tree. *random:* 1000,000 random points (diameter 141.421). *city:* 2,673,765 city coordinates (diameter 385.32). *covtype:* 581,012 data points with nine integer features from a tree coverage data set [Blackard and Dean, 1999] (diameter 10246.1).
- **pc** Point Correlation (number of points around a query coordinate within a radius) on a kd tree. Same datasets as *kd*.
- **mpc** Multi-Radius Point Correlation. The sample coordinate is drawn uniformly at random between 0.0 and q times the data box diameter.
- **km** k-means with 128 clusters (drawn uniformly at random from the data bounding box). Same datasets same as *kd*.
- **vp** 1-NN on a Vantage Point tree [Yianilos, 1993]. Same datasets as *kd*.
- **Binary tree (bt)** Element search on a binary tree. *random:* 16,777,216 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 [Tramm et al., 2014]. *random:* 10,000,000 elements.

Anatomy of Speculative Traversal. Figure 13.2 shows the data-parallel binary tree kernel (bt). We discuss it here as a prototypical example for all the tree traversal codes in the benchmark suite. When the kernel is vectorized the variable `i` is contiguous in the thread id (it has a $(1, W)$ shape).

The function maintains a stack that holds all nodes that need to be visited by at least one thread (Line 2). Each trip of the loop in Line 9 fetches a node off the stack. If any remaining thread in the loop is seeking an element that is less than the label of the current node, that node is pushed on the stack (Line 19). Symmetrically, the child node to the right is put on the stack (Line 21). If a thread finds the query element it takes a divergent exit out of the loop (Line 15). All threads start at the root of the tree. Threads diverge in the traversal if some threads proceed to the left child while others go to the right. In that case both nodes are put on the stack and when the child nodes are processed only a part of the threads actually make progress in the traversal. However, the later the threads diverge in their

```
int search(Node * nodes, float * Q, int i) {
  int stack[512];
  stack[0] = 0;
  int top = 1;

  float elem = Q[i];
  int result = -1;

  while (top > 0) {
    int next = stack[--top];
    float label = nodes[next].data;
    int right = nodes[next].right;
    int left = nodes[next].left;

    if (label == elem) {
      result = next;
      break;
    }
    if (any(mask() & (elem < label)) && left > 0)
      stack[top++] = left;
    if (any(mask() & (label < elem)) && right > 0)
      stack[top++] = right;
  }
  return result;
}
```

Figure 13.2.: Data-parallel element search in a binary tree.

descend into the tree the fewer child nodes are pushed on the stack and the quicker the stack is depleted. The efficiency of the traversal is thus input dependent, which is what makes this scheme speculative.

Programming Model. The *Clang* and *RV* versions of the tree traversal kernels are written as scalar C++ functions. The dataset dimension is a C++ template parameter. Each kernel version was instantiated four times: by a factor of two for the datatype (64 bit or 32 bit) in combination with the choice of dataset dimension (2D and 9D). The kernel versions for *RV* make use of horizontal operations (`popcount` and `any`) to implement the speculative traversal. The kernels are then whole-function vectorized by *RV* with pre-supplied vector shapes for the function arguments and into vector function signatures.

Multi-Radius Point Correlation. For the *bh*, *kd*, *pc*, *vp* and *km* benchmarks, the query coordinate is always *varying* while all other parameters to the query are uniform. It has been noted [Gray and Moore, 2001] 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.

ISPC versions. We re-implemented some of the *RV* kernels in the ISPC language. These are the 2D versions of *vp*,*pc*,*mpc*,*kd*,*kmeans* and *bintree*. We assist the ISPC compiler by exhaustively annotating variables and stack objects as `uniform` where possible. This means that the ISPC version of *mpc* is a complete reimplementation with different variables annotated as *uniform*. Note that annotation of variables is *not* necessary for the *RV* kernel versions since inference of *uniformity* is automatic, including the traversal stacks.

Query Inputs. The performance of speculative SIMD traversal codes is sensitive to how far the query inputs are apart in the search space. To this end, we evaluate some of the traversal codes with 10 different input sets that differ in their degree of coherence. These are *kd*, *vp*, *pc* and *mpc*. The query coherence is controlled by the *spread factor*, a real value between 0 and 1. For kernels with varying query coordinate (*kd*, *vp* and *pc*) the spread factor describes a bounding box for the query coordinates. The size of that bounding box in each dimension is fraction of the bounding box of the data set. For example, if the spread factor is 0.4, then the sample coordinates are drawn from a random box with 40% size in each dimension of the data bounding box. We draw 256 random coordinates from each sample box and for 10 query boxes in total. We do this to have the same query spread for all vector widths while guaranteeing that the query coordinates are identical across targets. The *pc* and *mpc* benchmarks have an additional radius argument. In *pc*, the sample radius is $1.0 - q$ times the diameter of the data box where q is the spread factor. The *mpc* uses q times the data bounding box diameter as the query radius. 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^{14} values uniformly at random from the dataset range. 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 compiler.

All tree benchmarks were evaluated in single threaded mode. Reported runtimes are the median execution times of the traversal parts in 35 full program iterations.

Note that our goal is generic vectorization of CFGs. Therefore, we do not compare against prior work on *dedicated* automatic vectorization of tree traversals [Jo et al., 2013] 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`.

13.1.1. Spread Factor Results

We evaluate the *pc*, *vp*, *kd* and *mpc* benchmarks with a spread factor ranging from 0.1 to 1.0 in increments of 0.1. The results for the 2D versions of those kernels are shown in Figure 13.3 for random coordinates and in Figure 13.4 for the *city* data set. The 9D version results for random coordinates are shown in Figure 13.5 and for the *covtype* data set in Figure 13.6.

We show runtime measurements as a function of the spread factor of the queries. We do so because the performance of speculative tree traversal kernels is sensitive to the spread of the query inputs. Comparing runtime measurements of a single query input is thus not representative for the performance behavior across the range of spreads.

This is because the kernels use a traversal stack to keep track of nodes that need to be visited in the traversal. This scheme is efficient if the query inputs grouped in the SIMD traversal function lie close together. Then, the set of all nodes that need to be visited for the whole of the query inputs is smaller than the sum of nodes that the scalar traversal needs to process. If the spread of the queries is high, the individual traversal of the input diverge early in the tree and the gain by speculative traversal is less pronounced. In the worst case, only one lane of the SIMD input will make progress in one iteration of the traversal loop.

Comparison to ISPC. Summarizing across all machines, kernels and configurations where ISPC is available, in the geometric mean RV delivers about 15% better performance compared to ISPC. There is a significant performance variation, however. In the worst case, ISPC is about 30% faster than RV (pc, 2D, rand, avx2). In the best case, RV is about 83% faster than ISPC (kd, 2D, rand, a53). RV is available with all configurations, different to the ISPC compiler, which only supports certain combinations of vector width, data type and target ISA.

Performance Impact of the Spread Factor. The pc kernel is different to kd, vp and mpc when it comes to the aspect of the spread factor. The sample radius is anti-proportional to the spread of the query coordinate. This configuration makes the performance of the pc kernel mostly invariant under

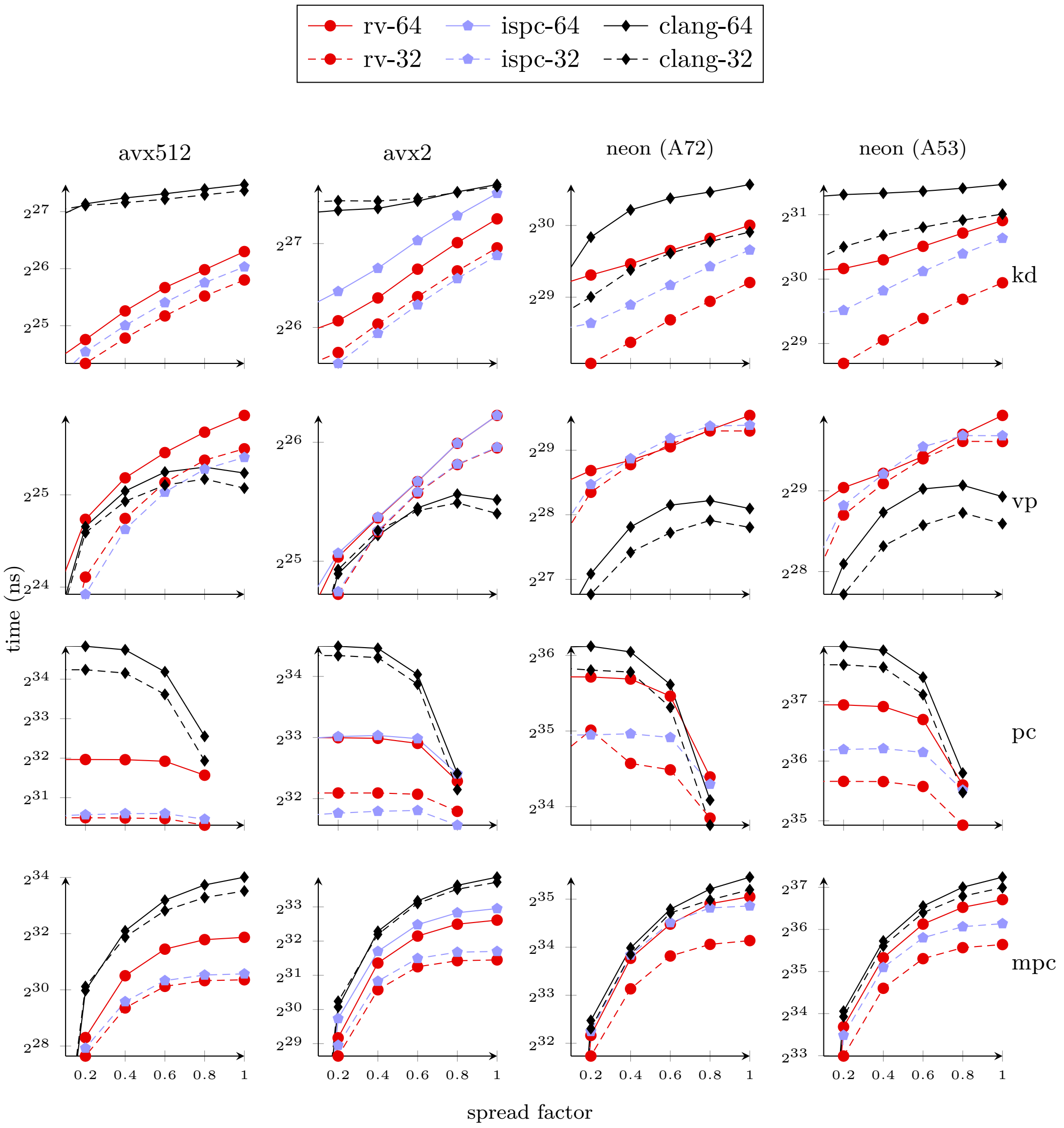


Figure 13.3.: 1-NN and PC kernels (2D random).

the spread factor. The spread factor 1 represents a degenerate corner case of the pc kernel and is thus left out from the plots. There, the query radius is 0 and since no coordinate lies within a sphere of size 0 the kernel does not perform any work. Regarding all other values of the spread factor for pc, we conclude that the performance hazard of spread out query coordinates is countered by the work saved due to a tighter query radius. Asymptotically, the scalar version scales better with the spread factor. The kd tree was build such that the leaf nodes hold up to 4 coordinates before they are subdivided and become inner nodes. The mpc kernel counts the number of coordinates within the query radius. The number of nodes that need to be visited are in the order of r^D where r is the query radius and D is the dimensionality of the data set.

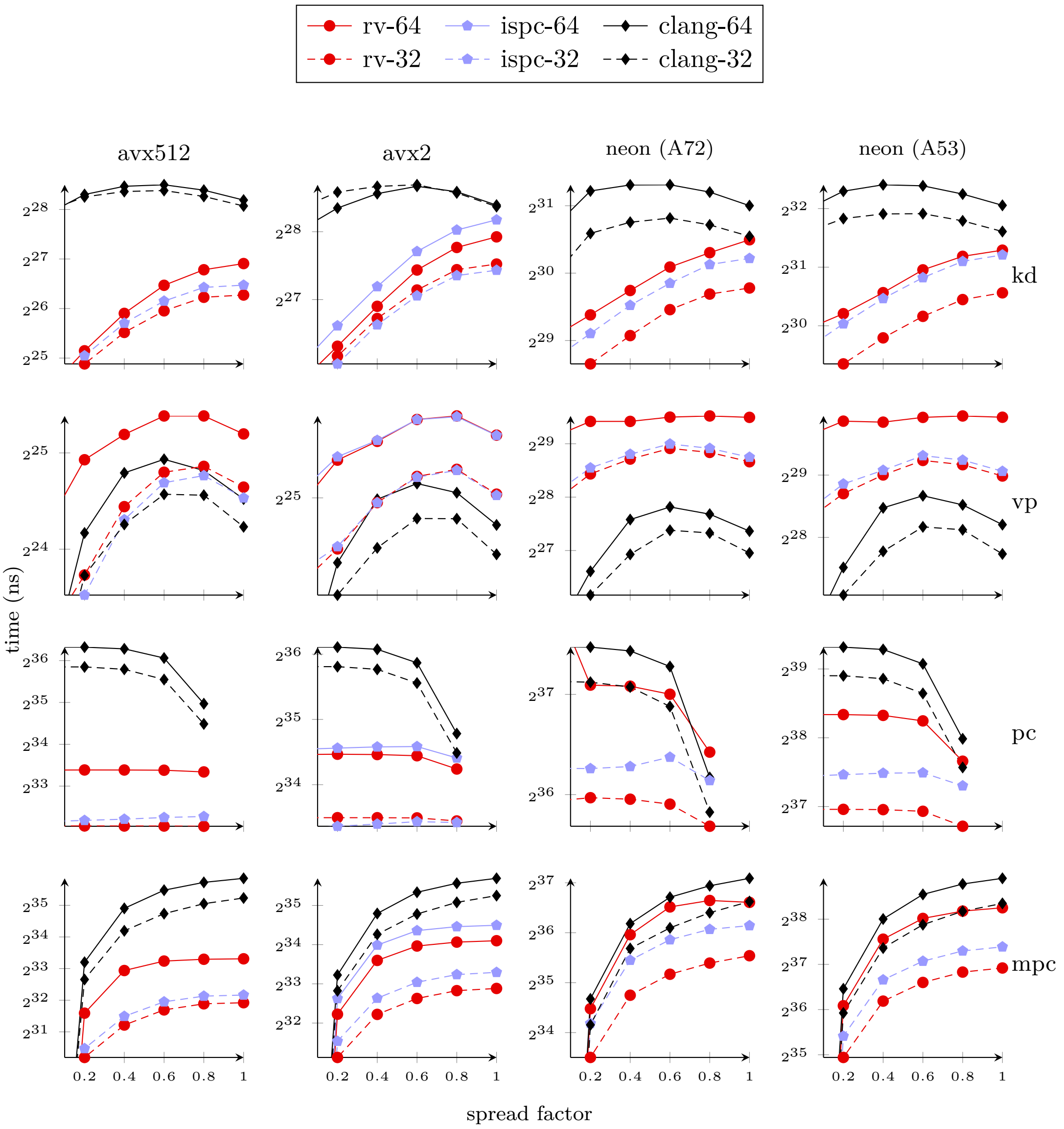


Figure 13.4.: 1-NN and PC kernels (2D city).

Dependence of Kernel Performance on the Data Set. Considering the 9d vp kernel, the random dataset favors the SIMD kernel whereas the random dataset is faster traversed with the scalar implementation. We observe that the performance of the 2D kernels is less susceptible to the choice of the data set.

Comparison of 64 bit against 32 bit Results. We see similar scaling behavior for 32 bit and 64 bit kernels for the same machine and data set. There is one outlier: the RV-vectorized version of the kd9 kernels takes a major performance hit compared to the scalar baseline.

The ISPC and RV kernels behave similar for all data sizes. Only for the 2D kd kernel, we see that in case of 64 bit, RV offers better performance than ISPC while it is the other way round for 32 bit. This is

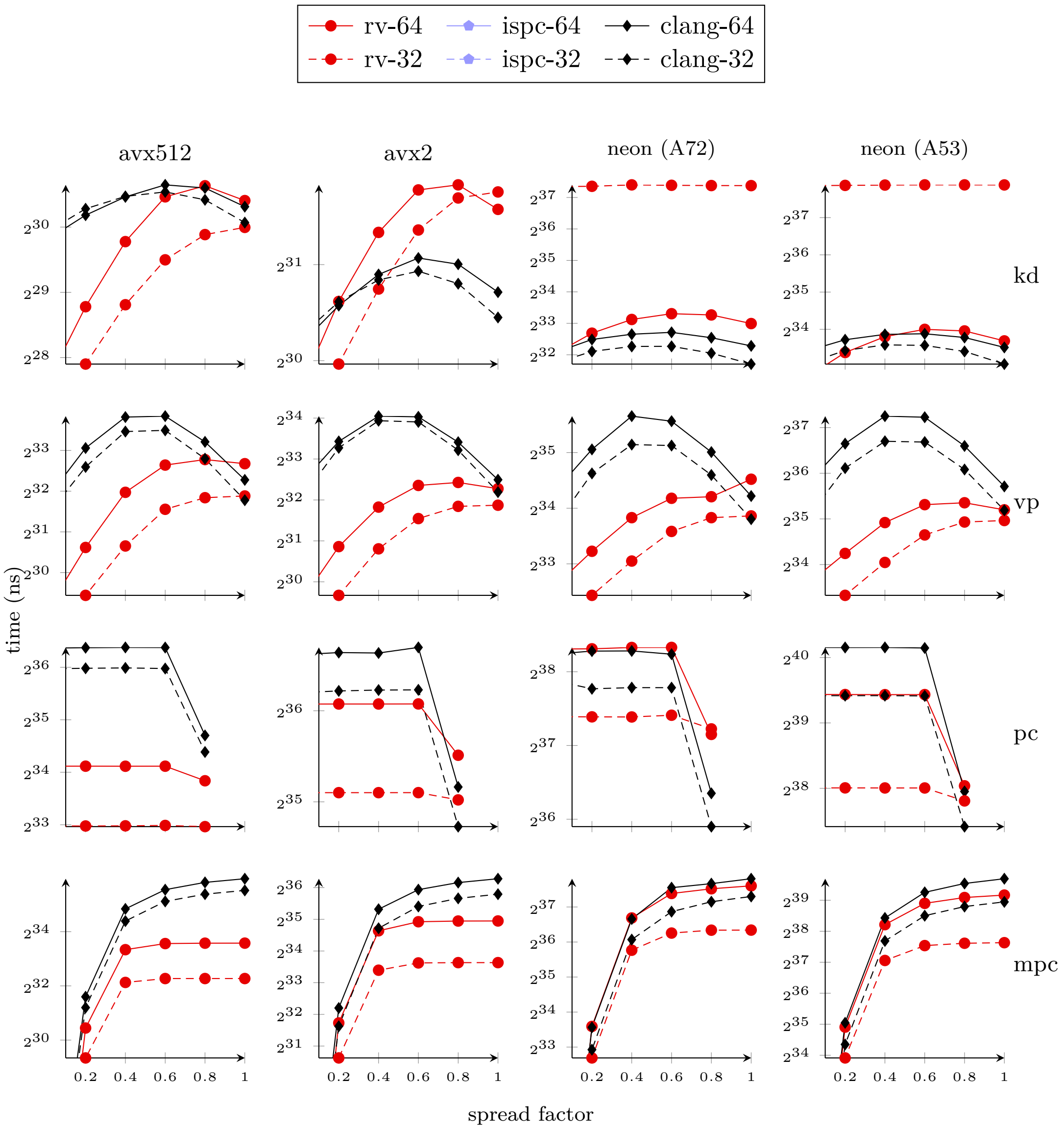


Figure 13.5.: 1-NN and PC kernels (9D random).

because for 32 bit the 2D coordinate type has 64 bit and fits exactly in the scalar register of the target machines. LLVM thus preserves these as `<2 x f32>` values in the IR. The kernel code however requires that the elements are extracted and broadcast, leading to inefficient SIMD code. The ISPC compiler breaks up the short vectors already in the frontend, an optimization that could also be implemented in RV.

Comparison against Scalar Baseline. The vectorized, speculative traversal scheme for traversal kernels is not always beneficial. For example, for an 1-NN search in the vantage point tree the scalar traversal is more efficient except for the 9D random dataset.

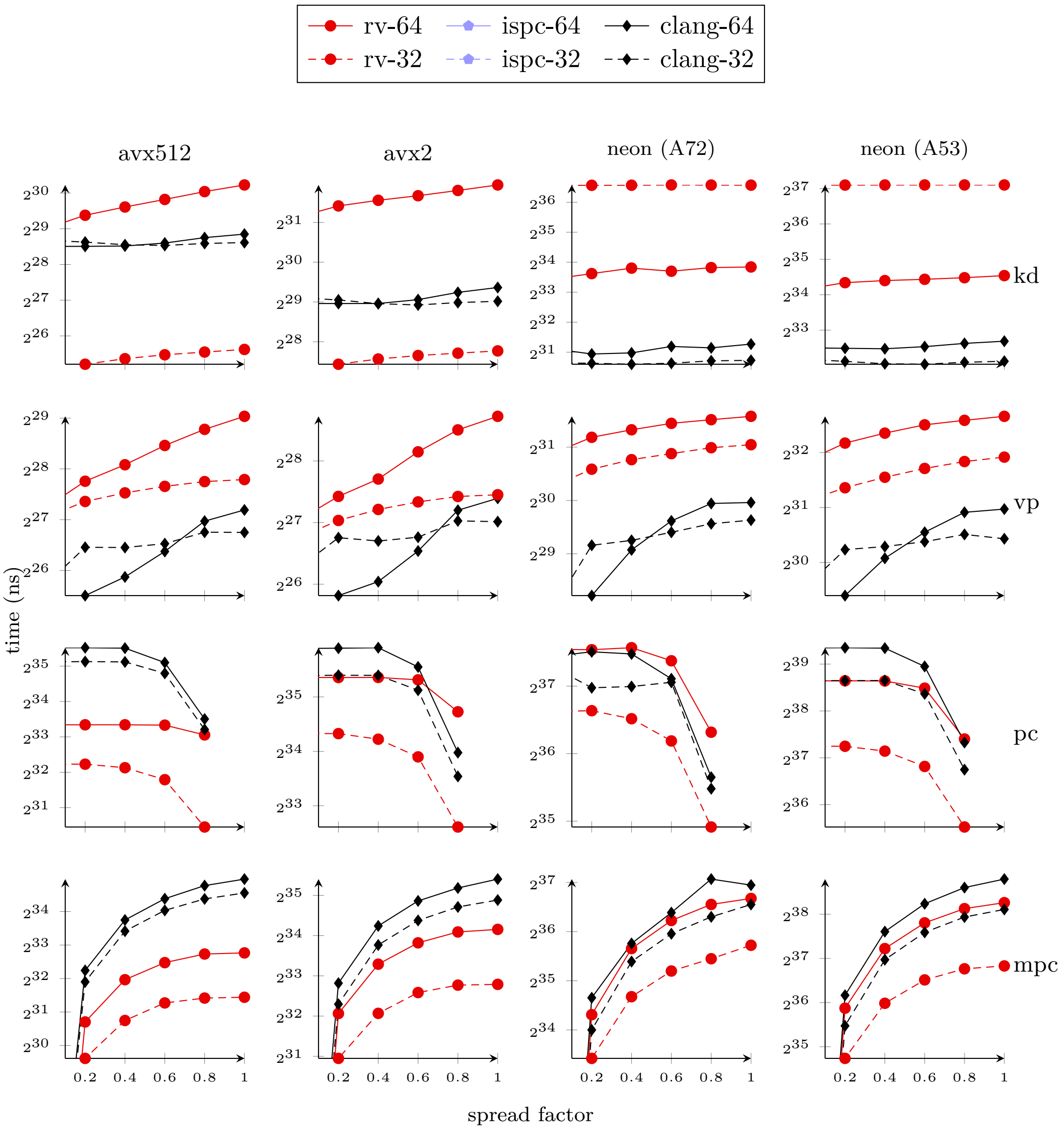


Figure 13.6.: 1-NN and PC kernels (9D covtype integer features).

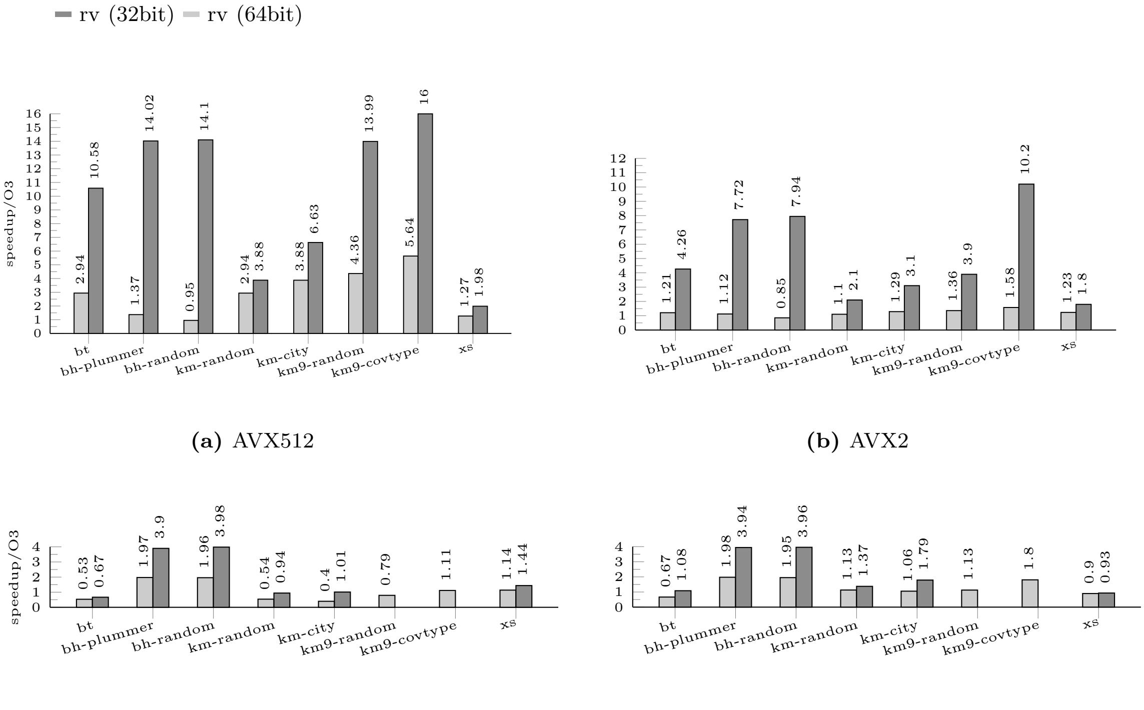
Divergence Analysis of Tree Traversal Codes. Divergence analysis statistics for the tree traversal codes discussed thus far are shown in the upper third of Figure 13.1.

As indicated in the *Allocas* column, the divergence analysis was able to prove that all traversal stacks are uniform. Proving stack uniformity exercises the Alloca SSA functionality of the divergence analysis Section 6.6.

Further, the 9D variants of the mpc,kd and pc kernels feature divergent loops, requiring the divergent loop transform Section 9.1. These loops are part of a box intersection test that loops over all dimensions and exits early if an intersection was found. That same loop is unrolled in the 2D variants of those kernels.

We show the vector shape of the activation masks separate for load and store operations. Partial linearization guarantees that a block with a uniform control mask will only execute if the mask is true (Section 8.6). Therefore, for uniform and constant masks, the loads and stores do not require masking at all. In the 9D case, loop divergence causes varying predicates and incurs masked memory accesses.

13.1.2. Other Tree Kernels



(a) AVX512

(b) AVX2

(c) A72

(d) A53

Figure 13.7.: Benchmarks without spread factor: Binary tree element search (bt), Barnes-Hut n-body simulation (bh), kmeans (km) and the quarry search kernel of the XSBench proxy application (xs).

Figure 13.7 shows runtime results on tree traversal benchmarks that do not have a spread factor. Due to code generation issues there is no kmeans9 result for 64 bit (`double`) on the AArch64 platforms. The binary tree implementation for RV is shown in Figure 13.2. The reported runtimes are normalized to the default Clang O3 pipeline. Increasing the vector width introduces more potential for divergence since more locations in the tree are accessed at once. However, comparing each 64 bit to each 32 bit result, the experiments show that doubling the vector width (halving the element size) is still beneficial.

13.2. Case Study: 644.nab_s

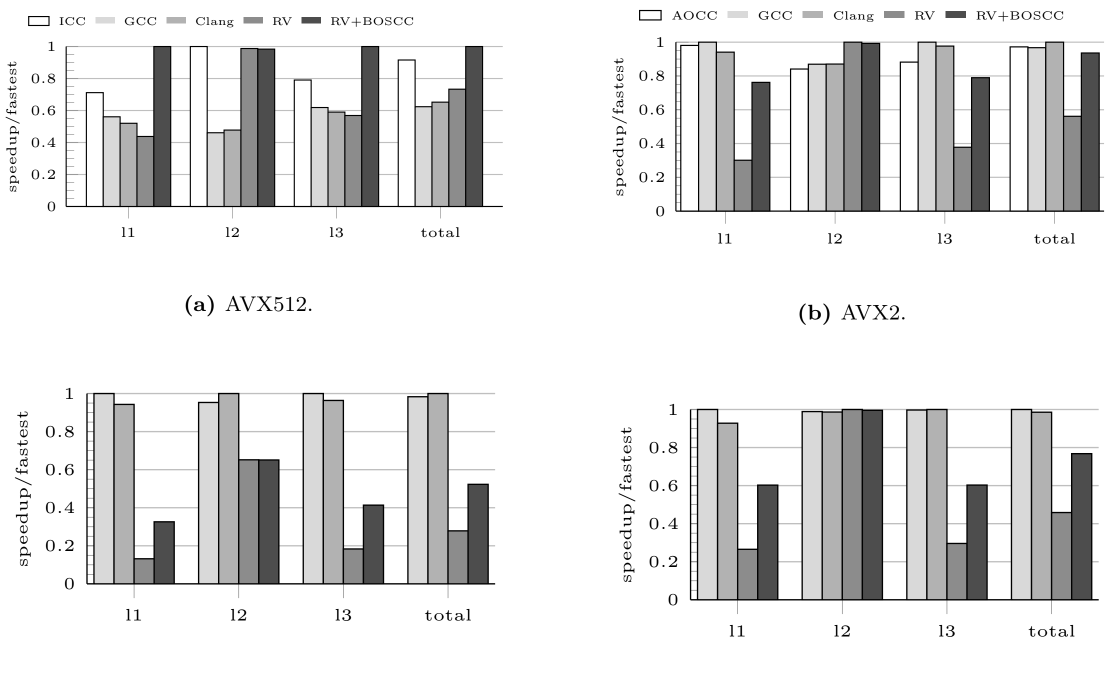
We use the `644.nab_s` benchmark of SPEC2017 to show the efficacy of the BOSCC gadget. We evaluated on the SPEC2017 *refspeed* data set for AVX512/AVX2 and on the *reftrain* data set for Adv. SIMD because of memory constraints. We compare against Clang (with PGO), GCC, ICC (Skylake-X only) and the AOCC (Ryzen2 only) compilers as shown in Figure 13.8a, Figure 13.8b, Figure 13.8d and Section 13.2.

About 77% of the running time in `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 by annotating them with the directive `#pragma omp simd` of OpenMP. 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 9.4.

The AOCC, GCC and Clang compilers are not able to vectorize any of the three loops.

RV inserts three BOSCC gadgets in each the first and the third loop. The BOSCC gadgets reflect in differences between the regular RV-vectorized variant and the RV+BOSCC variant in the code statistics table Figure 13.1. Code statistics for RV-vectorized loops without the gadget are named *nab/rv/n* where *n* is the number of the loop in program order. Statistics for the RV+BOSCC variants are analogously named *nab/boscc/n*.

Intel C Compiler. ICC chooses a vector width of 4 for the AVX512 Skylake-X target. RV vectorizes for the full vector width of 8 as the dominating data type is `double`. Inspection of the machine code reveals that ICC inserts a total of 16 BOSCC branches in the code. All three loops are vectorized. We use `-fp-model=precise` and add OpenMP reduction clauses to the loop pragmas to make sure that ICC performs FMA contraction but does not re-associate fp operations, except for the reductions in the SIMD loops.



(a) AVX512.

(b) AVX2.

(c) A72.

(d) A53.

Figure 13.8.: SPEC2017 - nab.

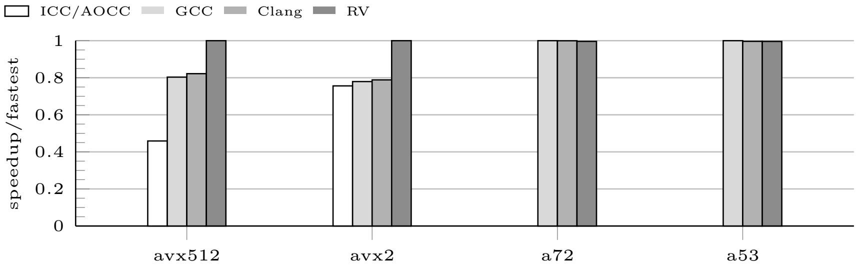


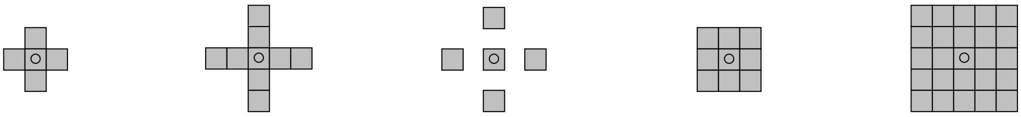
Figure 13.9.: XSBench results.

13.3. Case Study: XSBench

XSBench is a proxy benchmark for the *key computational kernel of the Monte Carlo neutronics application OpenMC [Romano et al., 2015].* About 85% of the total runtime of the actual OpenMC application is spent in this code [Tramm et al., 2014]. We run XSBench (Version 14) with the `nuclide` grid type option. The input sizes were *XL* for AVX2 and AVX512 and *large* for the ARM system due to memory constraints. We apply RV to an outer loop that internally runs the *xs* kernel as part of the simulation code. The loop is annotated with the `#pragma omp simd` of OpenMP for all tested compilers. As shown in Figure 13.1 the vectorization of that loop requires the divergent loop transform and partial linearization to preserve the uniform loop. The runtime results are shown in Figure 13.9. Our approach attains a speed up of 21.6% (AVX512) and 27.8% (AVX2) over the best of GCC, Clang and ICC. We did not observe any significant performance variation for the ARM system.

13.4. TensorRV: Stencil Codes

We evaluate the *TensorRV* system, described in Chapter 11, on five different stencil codes and matrix transpose. We show the stencil patterns of all evaluated stencils in Figure 13.10.



(a) Jacobi-5 **(b)** Jacobi-9 **(c)** Sparse Jacobi-5 **(d)** Seidel-9 **(e)** Seidel-25

Figure 13.10.: Stencils used in the evaluation. The central element (zero coordinate) of the stencils is marked with the ∘ symbol.

13.4.1. Experiment Setup

All results were obtained on the Skylake-X AVX512 machine. We evaluate every stencil with and without the modified data layout described in Section 11.7. The stencils are applied to an inner 1024 × 1024 patch of **`double`** data. The array boundaries are padded to accommodate the stencils (e.g. Jacobi-9 is evaluated on the interior of a 1026 × 1026 array).

13.4.2. Runtime Results

	2		4		8		16	
2			0.94	0.90	0.95	0.90	0.98	0.94
4	0.96	0.92	0.97	0.93	0.98	0.87		
8	0.82	1.03	0.92	0.93				
16	0.75	1.11						

(a) Jacobi-5

	2		4		8		16	
2			0.94	0.88	0.92	0.94	0.95	1.00
4	0.87	0.87	1.00	0.91	1.04	0.98		
8	0.70	0.99	0.91	1.04				
16	0.67	0.98						

(b) Seidel-9

	2		4		8		16	
2			1.04	1.16	0.95	1.02	1.03	1.12
4	0.90	1.18	1.09	1.19	1.02	1.15		
8	0.78	1.25	1.05	1.23				
16	0.74	1.24						

(c) Jacobi-9

	2		4		8		16	
2			0.93	0.92	1.19	1.10	1.10	1.03
4	0.65	0.85	1.01	0.96	1.29	1.20		
8	0.56	0.80	0.89	1.01				
16	0.49	0.72						

(d) Seidel-25

	2		4		8		16	
2			0.96	1.05	0.94	0.96	0.97	1.03
4	0.96	1.06	0.95	1.09	1.02	1.06		
8	0.96	1.15	0.99	1.16				
16	0.90	1.26						

(e) Sparse Jacobi-5

	2	4	8	16
2		1.59	1.31	1.26
4	2.30	2.37	2.00	
8	3.79	3.78		
16	3.92			

(f) Matrix transpose

Figure 13.11.: Single core TensorRV results. **Columns:** inner-loop brush size. **Rows:** outer-loop brush size. **Left sub columns:** unchanged data layout. **Right sub columns:** with 2 × 2 data layout.

We show single core results in Figure 13.11 and multi core results in Figure 13.12. The reported numbers are speedups over vectorization in the inner-most loop with a vector length of 8, which is the native vector length for **`double`** elements on AVX512. We report median results of 51 iterations. We explain the structure of the runtime tables with an example. Consider the 1.23 result reported in the

	2		4		8		16	
2			1.00	1.00	0.96	1.04	0.99	1.25
4	0.96	1.19	0.92	1.23	0.95	1.15		
8	0.91	1.15	0.85	1.06				
16	0.79	1.14						

(a) Jacobi-5

	2		4		8		16	
2			0.86	1.09	0.99	1.23	1.02	1.10
4	0.87	1.28	0.96	1.07	0.99	1.08		
8	0.82	0.99	0.90	1.23				
16	0.85	1.04						

(b) Seidel-9

	2		4		8		16	
2			0.97	1.35	1.00	1.11	0.95	1.11
4	0.97	1.09	0.96	1.40	0.97	1.46		
8	0.83	1.40	0.94	1.39				
16	0.82	1.10						

(c) Jacobi-9

	2		4		8		16	
2			0.92	1.07	0.99	1.02	1.01	1.10
4	0.76	0.85	1.02	1.05	1.02	1.35		
8	0.79	1.01	0.92	1.04				
16	0.58	0.84						

(d) Seidel-25

	2		4		8		16	
2			1.01	1.20	0.97	1.14	0.99	1.13
4	0.97	1.07	0.93	1.14	0.97	1.46		
8	0.92	1.28	0.93	1.30				
16	0.91	1.06						

(e) Sparse Jacobi-5

	2	4	8	16
2		1.26	1.31	1.26
4	1.90	1.97	1.48	
8	6.03	6.07		
16	6.11			

(f) Matrix transpose

Figure 13.12.: Multi core TensorRV results. Result tables structured as in Figure 13.11.

fourth column and third row of the table in Figure 13.11c. The runtime table has row labels and column labels, which tell us the tensor brush that was used in this configuration. The column indicates the size of the brush in the inner loop, in this case 4. The row tells us the brush size in the outer loop, in this case 8.

The are two entries in the table for that brush, of which the 1.23 result is on the right. The left entry is the speedup without data-layout transformation and the right one was measured with the 2×2 layout. We thus know that the 9-point Jacobi stencil with a 2×2 data layout and the brush $\mathcal{B} = 8 \times 4$ achieves a speedup of 1.23 over the loop vectorized version.

13.4.3. Machine Code Statistics

We complement the runtime results with an analysis of the machine code of each stencil variant.

The result of this is analysis is shown in Figure 13.13. The AVX512 SIMD instructions in the assembly fall into three categories: shuffle instructions, memory instructions and floating-point instructions. According to Agner Fog's instruction tables [Fog, 2011], the execution ports for these instructions mostly do not interfere.

Shuffle instructions (shuffle ops) are those matching `vshuff*` or `vperm*` or `vinsert*` (port 5). Floating-point instructions (fp ops) are those matching `vadd*` or `vmul*` (port 0 or 5). Memory instructions (memory ops) are those matching `vmov*` with an address operand (ports 2,3,7,4). Shuffle instructions that combine a load and a shuffle operation are counted twice as a shuffle op and a memory op. [1]

Each one of the three scatter plots in Figure 13.13 plots the relative amount of one instruction category (y axis) against the speedup over the inner-loop vectorized version (x axis). Every data point is one combination of stencil, brush, data layout and threading configuration. This includes the base line versions with a speedup of 1.0.

[1] The code contains a few other, scalar instructions, e.g. for loop control. The Skylake-X CPU executes those in parallel with the AVX512 instructions and on separate execution units. Hence, scalar instructions can be ignored for the purposes of performance analysis.

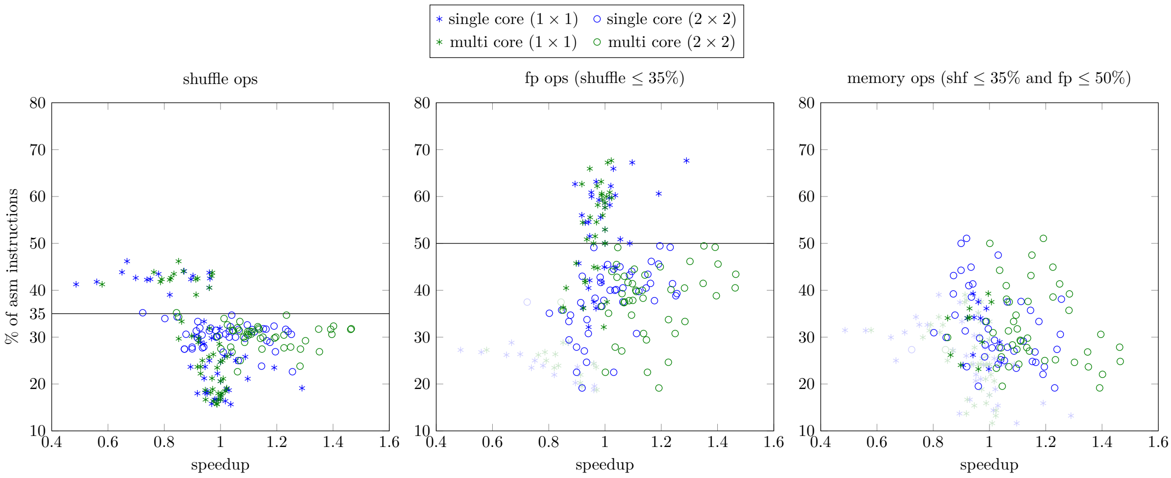


Figure 13.13.: Each data point is one combination of stencil (excluding transpose), datalayout, brush and threading. Speedup over inner-loop vectorized version (x axis). Left: fraction of shuffle instructions. Center: fraction of fp arithmetic instructions. Right: fraction of memory instructions.

Threshold Lines. The data points in the first plot form two clusters. We overlay the plot with a threshold line at 35% shuffle instructions to separate the clusters. Data points above that threshold are drawn in light colors in the second and third plot. We apply another threshold in the second plot at 50% floating-point instructions. Data points that are above that threshold are again drawn in light colors in the third plot on the right-hand side. Thus, if a data point in the third plot is drawn with a light color then the corresponding stencil variant surpasses at least one of the thresholds in the plots to the left.

13.4.4. Discussion

We start with observations on all stencil variants before moving into the discussion of individual stencils. These observations on stencils do not apply to matrix transpose because it has a different memory access pattern and no computations.

Interpretation of Machine Code Statistics. We turn to the machine code statistics in Figure 13.13 to get an overview over all generated stencil variants.

Some variants on the standard layout are imbalanced in that their machine code is dense in shuffle instructions. Those are the variants in the upper cluster in the shuffle ops plot of Figure 13.13 with 35% shuffle instructions and above. The Skylake-X CPU can issue up to one shuffle instruction in each cycle [Fog, 2011]. We interpret the slowdown of the variants in the first cluster as a sign that this capacity is maxed out at around 35% shuffle instructions.

As shown in the center plof of Figure 13.13, variants that have more than 50% floating-point operations are below a speedup of 1.1× with the exception of two outliers. These are two variants of the Seidel-25 stencil with a 2×8 and 4×8 brush both on the 1×1 data layout.

Effect of Data Layout Transformation. The thresholds in the first and second plot of Figure 13.13 exclusively separate off variants of the standard layout. Stencil variants with the transformed layout are more balanced in their mix of SIMD instructions. This results in better performance as shown in Figure 13.11 and Figure 13.12.

We compare the runtimes in Figure 13.11 and Figure 13.12. We find that switching from single to multi core execution generally amplifies the speedups. This holds in particular for the 2×2 data layout variants that already surpass the performance of the inner-most vectorized baseline. There are also improvements for the standard layout when switching to multi core execution. However, those stay below the performance of the baseline.

Matrix Transpose. The matrix transpose kernel is the operation `A(i,j) = B(j,i)` where `i` is the iteration variable of the inner loop. The store to `A` is therefore contiguous in the iteration variable `j` of the outer-loop. Consider the performance of the kernel for different brush sizes in Figure 13.12. The speedup increases with the brush size in the outer loop. The exact cause for this performance behavior merits further investigation.

13.5. Conclusion

How does the RV vectorization system compare to other data-parallel vectorizers? Our comparison against ISPC shows that RV is able to match and sometimes outperform it. However, RV is a much more flexible system vectorizing any LLVM IR program with reducible control flow whereas ISPC is strictly limited to its structured DSL representation. The tree traversal codes were manually translated into the ISPC language. This required extensive use of `uniform` qualifiers to compensate for deficiencies in the divergence analysis of ISPC and reach the same SIMD code quality as RV. In comparison, RV vectorized data-parallel C++ implementations where only the vector shapes of function arguments were pre-supplied.

How relevant is RV for actual application codes? We conducted two case studies to see how RV performs in standard benchmark codes. Our results show that RV is able to achieve speedups over state-of-the-art C++ compilers for those platforms. Different to the tree traversal results, these benchmarks were not written for automatic vectorization. The results thus corroborate that the transformations for divergent control flow offered by RV are also beneficial in general applications codes.

What are the performance benefits of tensorization and how performance-sensitive is the technique to the brush? The TensorRV results show that the framework is capable of delivering speedups on stencil codes over naive loop vectorization. However, tensorization extends the search space: which brush and data layout should be used for which kernel? Our experiments show that there is not one best brush size but that this depends on the stencil at hand. Still, we found some hard thresholds for performance in the fraction of shuffle and floating point operations. We leave the exploration of cost factors and the elaboration of a proper cost model for TensorRV to future work.

Chapter 14.

Conclusions

Technological limitations do not admit to continue the trend of ever-improving scalar compute performance. Instead CPUs offer significant parallel processing power locked in their SIMD units. Exploiting SIMD is thus mandatory to implement high-performance applications on modern CPUs. There are two main ways to program SIMD: either directly through compiler builtin functions and types, which are tedious to program and result in target-specific code, or through vectorization tools and libraries.

Vectorization of data-parallel codes is a fundamental technique that underlies many of these tools, ranging from outer-loop vectorizers to whole-function vectorizers to vectorizers for explicitly data-parallel programming languages. While vectorization techniques for structured programs are well understood and readily available, there was no reliable vectorization framework for unstructured codes. Yet, many programs are naturally unstructured or become unstructured due to beneficial code optimizations.

This thesis presents the RV vectorization system for data-parallel, unstructured codes in a CFG-based compiler IR. The RV system features a novel vectorizer IR along with algorithms and analyses to build a complete vectorizer system from it. RV vectorizes all IR programs with reducible control flow that are valid under data-parallel, lock-step execution.

We observe the following: First, a general, data-parallel vectorizer for unstructured CFGs can match and exceed the performance of more specialized systems for structured languages or outer-loop vectorization. Second, the developed algorithms are surprisingly simple, making feasible proofs of correctness and other properties. Proved properties simplify transformations or enable more aggressive optimizations. Finally, by defining a single, full-featured vectorizer IR for the whole vectorization process, vectorization becomes a question of composing transformation passes, a model that has proven itself in modern compilers.

We believe that the RV system offers a robust foundation for future research in data-parallel vectorization. The exact relation between greedy schedules and partial linearization warrants further investigation and the space of runtime schedules is largely unexplored. We also leave to future work the extension to irreducible control-flow, which prompts the question which iterations in irreducible loops are supposed to synchronize.

On the practical side, we see opportunities in inter-procedural vectorization and novel transformations that were too complicated to implement in earlier systems.

Appendix A.

Correctness and Precision of *local_joins*

Listing 9 is a simplified variant of the *local_joins* algorithm in Listing 4 of Chapter 7. The claims of correctness and precision are enshrined in Theorem 2, which we prove in the conclusion of this appendix.

Listing 9: Single-source, all-sinks disjoint paths algorithm for DAGs.

```
Input: $A \in \mathtt{V}$, Divergent branch label.
Output: $DomMap$ : dominating node map.
$DomMap \leftarrow \{b \rightarrow \bot \mid b \in \mathtt{V}\}$
// Divergent Successors
foreach divergent $A \rightarrow s$ do
    $DomMap[s] \leftarrow s$
end
// Uniform Successors
foreach uniform  $A \rightarrow s$ do
    $DomMap[s] \leftarrow \mathbf{u}$
end
// Join computation
foreach $b$ in $rpo()$ do
    $d \leftarrow DomMap[b]$
    if $d = \bot$ then
        foreach $q \rightarrow b$ do
            $d \leftarrow d \overset{b}{\sqcup} DomMap[q]$
        end
        $DomMap[b] \leftarrow d$
    end
end
```

Definition 25 (Disjoint Path Join)**.**

$$
\begin{array}{rcl}
\overset{x}{\sqcup} : \mathtt{V}' \times \mathtt{V}' & \rightarrow & \mathtt{V}' \\
a \overset{x}{\sqcup} b & \mapsto & \begin{cases} a & \textit{if } a = b \vee b = \bot \\ b & \textit{if } a = \bot \wedge b \neq \bot \\ x & \textit{otherwise} \end{cases}
\end{array}
$$

where $\mathtt{V}' = \{\, \mathbf{u}, \bot \,\} \cup \mathtt{V}$.

We make the following assumptions to simplify the proofs. We assume for every successor s of A that A is the only predecessor of s, that is $A \to s$ is the only edge leading into s. This property can be established by replacing every edge $(A, i, s) \in \mathtt{E}$ with two edges (A, i, p_A) and $(p_A, 0, s)$ for a new, unique block p_A with an unconditional branch to s. This operation takes time linear in the number of original control-flow edges. Further, as established in Section 2.4, every exit block of a loop has exactly one incoming loop-exiting edge.

Normally, either branches are completely uniform (there is no control divergence) or they are completely divergent (disjoint paths may originate from any control-flow edge out of the node). Partial divergence arises from collapsing divergent loops into nodes with some uniform and some divergent successors as described in Section 7.3.

Lemma 3 (Properties of the Disjoint Path Join)**.** *The Disjoint Path Join operator* $\overset{x}{\sqcup}$ *is distributive, commutative, and associative.*

Lemma 4 (Monotonicity of ***DomMap***)**.** *For any* $b \in \mathtt{V}$ *either* $\mathit{DomMap}[b] \in \{\perp, \mathbf{u}\}$ *or* $\mathit{BlockIndex}(\mathit{DomMap}[b]) \leq \mathit{BlockIndex}(b)$*.*

Lemma 5 (Completeness of Listing 9)**.** *Let* $A \in \mathtt{V}$*, then for the result of Listing 9, the following are equivalent for any* $p \in \mathtt{V}, \mathit{BlockIndex}(p) \geq \mathit{BlockIndex}(A)$

1. $\mathit{DomMap}[p] \neq \perp$*.*
2. *There exists a path* $A \to^+ p$*.*

Proof. We prove the claim by induction over the block index. The induction hypothesis is Lemma 5, where $p \in \mathtt{V}$ is the induction variable, identified by its block index number $n \in \mathbb{N}$, respectively.

Base case ($\mathit{BlockIndex}(p) = 0$) The graph is acyclic and so there is no edge $p \to p$. Hence, $\mathit{DomMap}[p] = \perp$. Also, there can be no path from p to p in a DAG.

Step ($n \to n+1$, direction (1) $\implies$ (2)) Let $\mathit{BlockIndex}(p) = n + 1$ with $\mathit{DomMap}[p] \neq \perp$. There are two ways that Listing 9 can define $\mathit{DomMap}[p]$ to be different from $\perp$: Either, p is an immediate successor of A and so $A \to p$ is a path from A. Otherwise, $\mathit{DomMap}[p]$ is computed as $\overset{p}{\bigsqcup_{q \to p}} \mathit{DomMap}[q]$. By Definition 25, for $\mathit{DomMap}[p] \neq \perp$ there has to be a $q \to p$ such that $\mathit{DomMap}[q] \neq \perp$. We apply the induction hypothesis for $\mathit{BlockIndex}(q) < \mathit{BlockIndex}(p)$ with $\mathit{DomMap}[q] \neq \perp$ and get that there is a path $A \to^* q$. Since $q \to p$, also $A \to^+ p$.

Step ($n \to n+1$, direction (2) $\implies$ (1)) Let $\mathit{BlockIndex}(p) = n + 1$ such that there is a path $t \in A \to^* q \to p$ for some $q \in \mathtt{V}$. Consider the case $q = A$, then $A \to p$ and $\mathit{DomMap}[p] \in \{\mathbf{u}, p\}$ and the claim follows directly. Otherwise, $t \in A \to^+ q \to p$ and t has a prefix path $r \in A \to^+ q$. In this case, p is not the immediate successor of A and $\mathit{DomMap}[p]$ is computed as $\overset{p}{\bigsqcup_{q \to p}} \mathit{DomMap}[q]$ Since $\mathit{BlockIndex}(q) < \mathit{BlockIndex}(p)$ and $r \in A \to^+ q$, the induction hypothesis yields $\mathit{DomMap}[q] \neq \perp$.

Hence, also $\mathit{DomMap}[p] \neq \bot$.

□

Lemma 6 (Divergent path out of $\boldsymbol{A}$)**.** *Given any $A, p \in \mathbb{V}$. The following statements are equivalent:*

1. *There is a path $A \to q \to^* p$ such that $A \to q$ is a divergent edge out of A.*

2. *$\mathit{DomMap}[p] \notin \{\, \bot, \mathbf{u} \,\}$.*

Proof. We prove the claim by induction over the length of the longest path from $A \to^+ p$.

Base case ($A \to p$), (1) $\implies$ (2) Let $A \to p$ be a divergent edge out of A. Since p is an immediate divergent successor of p, $\mathit{DomMap}[p] = p \notin \{\, \bot, \mathbf{u} \,\}$.

Base case ($A \to p$), (2) $\implies$ (1) Let $\mathit{DomMap}[p] \notin \{\, \bot, \mathbf{u} \,\}$. By Definition 25 and since A is the only predecessor of p, it must hold that $A \to p$ is a divergent edge out of A.

Step (longest path length $n \geq 2$), (1) $\implies$ (2) There is a simple path $A \to^+ q \to p$ that starts in a divergent edge. We apply the induction hypothesis to q and obtain that $\mathit{DomMap}[q] \notin \{\, \mathbf{u}, \bot \,\}$. Being a successor of $q \neq A$, p is not also an immediate successor of A. Therefore, $\mathit{DomMap}[p]$ is computed as $\mathit{DomMap}[p] = \mathit{DomMap}[q] \overset{p}{\sqcup} X$ for X representing the join of all other predecessors of p. By Definition 25 and because $\mathit{DomMap}[q] \notin \{\, \mathbf{u}, \bot \,\}$, $\mathit{DomMap}[p] \notin \{\, \mathbf{u}, \bot \,\}$.

Step (longest path length $n \geq 2$), (2) $\implies$ (1) The node p is not an immediate succssor of A. Therefore, $\mathit{DomMap}[p]$ is computed as $\mathit{DomMap}[p] = \overset{p}{\bigsqcup_{q_i \to p}} \mathit{DomMap}[q_i]$. Given that $\mathit{DomMap}[p] \notin \{\, \mathbf{u}, \bot \,\}$ and Definition 25 there has to be such a $q_i \to p$ with $\mathit{DomMap}[q_i] \notin \{\, \mathbf{u}, \bot \,\}$. The longest path to that q_i is shorter than the longest path to p and we can apply the induction hypothesis. This yields a path $t \in A \to^+ q$ that starts in a divergent edge. The extension $t \to p$ also starts in a divergent edge.

□

Corollary 1. *Given that there exists a path $A \to^+ p$. $\mathit{DomMap}[p] = \mathbf{u}$, only if for all paths $t \in A \to q \to^* p$ for any q it holds that $A \to q$ is a uniform edge out of A*

Proof. This is the contraposition of Lemma 6. □

Theorem 2 (Disjoint Path Invariant)**.** *Let DomMap be the result of Listing 9 for a partially divergent node in label A. Consider any two $p, p' \in \mathbb{V}$. If $\mathit{DomMap}[p] = d$ and $\mathit{DomMap}[p'] = d'$ with $d, d' \neq \bot$, then the following two statements are equivalent*

1. *$d \neq d'$*

2. *There exist almost node-disjoint paths* $t \in A \to q \to^* p, t' \in A \to q' \to^* p'$ *such that at least one of* $A \to q$ *or* $A \to q'$ *is a divergent edge out of* A.

Proof. First, consider the case that $p = p'$. Trivially, $\mathit{DomMap}[p] = \mathit{DomMap}[p']$ and there are also not two almost node-disjoint paths $t \in A \to^+ p$ and $t' \in A \to^+ p$. For the following proof, we may therefore assume that $p \neq p'$ and wlog that $\mathit{BlockIndex}(p) < \mathit{BlockIndex}(p')$

We will proof Theorem 2 by induction over the block index number $n \in \mathbb{N}$.

Induction Hypothesis
Theorem 2 holds for any two nodes $p, p' \in \mathbb{V}$, such that $\mathit{BlockIndex}(p) \leq n$ and $\mathit{BlockIndex}(p') \leq n$.

Base case ($n = 0$)
Trivially, there are no two labels $p, p' \in \mathbb{V}$ with $p \neq p'$ and $\mathit{BlockIndex}(p) \leq 0$ and $\mathit{BlockIndex}(p') \leq 0$.

Base case ($\mathit{DomMap}[p'] = \bot$)
By Lemma 5, there is no path $A \to^+ p'$.

Base case ($A \to p'$ and $\mathit{DomMap}[p'] = p'$)
Given any $p \in \mathbb{V}$ with $\mathit{BlockIndex}(p) < \mathit{BlockIndex}(p')$ it is clear that for any path $t \in A \to^+ p$, it holds that $p' \notin t$ because the graph is acyclic. Also $\mathit{DomMap}[p] \neq p'$ because of Lemma 4.

Base case ($A \to p'$ and $\mathit{DomMap}[p'] = \mathbf{u}$)
Consider the case that $\mathit{DomMap}[p] = \mathbf{u}$. By Corollary 1, equivalently all paths $A \to^+ p$ therefore start in a uniform edge and there is no path $A \to^+ p$ that starts in a divergent edge. Otherwise, $\mathit{DomMap}[p] \notin \{\, \mathbf{u}, \bot \,\}$. By Lemma 6, equivalently there has to be a path $A \to^+ p$ starting in a divergent edge out of A. Given that for any path $t \in A \to^+ p$, it holds that $p' \notin t$ because the graph is acyclic.

Induction Step ($n \to n + 1$), (1) $\implies$ (2)
Case ($d' = \mathbf{u}$ and $d \neq \mathbf{u}$)
Given $d' = \mathbf{u}$, all paths $A \to^+ p'$ are uniform and do not start in start in a divergent edge. Since this is covered by a base case, we may assume that there is no edge $A \to p'$. Therefore for all edges $q' \to p'$, it must hold that $\mathit{DomMap}[q'] \in \{\, \mathbf{u}, \bot \,\}$. Otherwise not $d' = \mathbf{u}$ because of Definition 25. By the same token, there has be an $q' \to p$ such that $\mathit{DomMap}[q'] = \mathbf{u}$. As $d \notin \{\, \mathbf{u}, \bot \,\}$ and $\mathit{DomMap}[q'] = \mathbf{u}$, we can apply the induction hypothesis. By the IH, there exist two almost node-disjoint paths $t'_q \in A \to^+ q'$ and $t \in A \to^+ p$ such that one of them starts in a divergent edge out of A. Note that since $\mathit{DomMap}[q'] = \mathbf{u}$ and by Corollary 1, among the two paths only t can start in a divergent edge out of A. Also, the extension $t'_q \to p'$ is also almost node-disjoint from t because $\mathit{BlockIndex}(p) < \mathit{BlockIndex}(p')$ and the

graph is acyclic.

Induction Step ($n \to n + 1$), (1) $\implies$ (2)
Case ($d' \neq \mathbf{u}$)
Assume $d \neq d'$. Since this is covered by a base case, we may assume that there is no edge $A \to p'$. Then, there has to be a $q' \to p'$ such that $\mathit{DomMap}[q'] \neq \mathbf{u}$. This is because otherwise by Definition 25 $\mathit{DomMap}[p'] = \mathbf{u}$, which violates the assumption of the case.

Further, either $\mathit{DomMap}[q'] \neq d$ or $\mathit{DomMap}[q'] = d$ and there has to be another $q'' \to p'$ such that $\mathit{DomMap}[q''] \notin \{\, \bot, d \,\}$, also by Definition 25. In either case, there has to be a predecessor $q''' \to p$ with $\mathit{DomMap}[q'''] \notin \{\, \bot, d \,\}$. We can apply the induction hypothesis to q''' and p: There are two almost node-disjoint paths $t_q \in A \to^+ q'''$ and $t \in A \to^+ p$ and one of them starts in a divergent edge out of A. Then, again, the extension $t' = t_q \to p'$ is also almost node-disjoint from t. If t was the divergent path out of A, then the claim follows directly. Otherwise if t_q is a divergent path out of A, then so is its extension t'.

Induction Step ($n \to n + 1$), (2) $\implies$ (1)
Case (almost node-disjoint paths $t \in A \to^+ p$ and $t' \in A \to^+ p'$, t is divergent out of A)
Since $t' = A \to p'$ is covered by a base case, we can assume that $t' = A \to^+ q' \to p'$. By Lemma 5 and the existence of t and t', $\bot \notin \{\, \mathit{DomMap}[p], \mathit{DomMap}[p'], \mathit{DomMap}[q'] \,\}$. Further since t is a divergent path out of A, $\mathit{DomMap}[p] \neq \mathbf{u}$.

We will assume the contradiction: $\mathit{DomMap}[p'] = \mathit{DomMap}[p]$. From Lemma 4, we conclude that $\mathit{DomMap}[p'] \neq p'$ because $\mathit{DomMap}[p] \neq d \notin \{\, \mathbf{u}, \bot \,\}$. Since $\mathit{DomMap}[q'] \neq \bot$, it must also hold that $\mathit{DomMap}[q'] = p'$. However, the prefix $A \to^+ q'$ is almost node-disjoint from the path t and hence the induction hypothesis yields $\mathit{DomMap}[q'] \neq \mathit{DomMap}[p]$. This is a contradiction since not at the same time $\mathit{DomMap}[q'] \neq \mathit{DomMap}[p]$ and $\mathit{DomMap}[q'] = \mathit{DomMap}[p'] = \mathit{DomMap}[p]$. Therefore, $\mathit{DomMap}[p'] \neq \mathit{DomMap}[p]$.

Induction Step ($n \to n + 1$), (2) $\implies$ (1)
Case (almost node-disjoint paths $t \in A \to^+ p$ and $t' \in A \to^+ p'$, t' is divergent out of A)

$\mathit{DomMap}[p'] \notin \{\, \mathbf{u}, \bot \,\}$ because of Lemma 5 and Lemma 6. Also, $\mathit{DomMap}[q] \neq \bot$ because of Lemma 5. Since $t' = A \to p'$ is covered by a base case, we can assume that $t = t_q \to p'$ with $t_q \in A \to^+ q'$. Since t_1 is a prefix of t' it also does not start in a divergent edge out of A. The induction hypothesis thus yields that $\mathit{DomMap}[q'] \neq \mathit{DomMap}[p]$.

However, as p' is not a direct successor of A and $q' \to p'$ the value $\mathit{DomMap}[p'] = \mathit{DomMap}[q'] \overset{p'}{\sqcup} X$ for some $X \in \mathbb{V}'$. Then, $\mathit{DomMap}[p'] \neq \mathit{DomMap}[p]$ because $\mathit{DomMap}[q'] \neq \mathit{DomMap}[p]$.

□

Theorem 7 (Sync Dependence Criterion)**.** *Given $A, z \in \mathbb{V}$, the following statements are equivalent:*

1. *There exist two edge-disjoint paths $t, t' \in A \to^{+} z$ such that at least one of them starts in a divergent edge ouf of A and t and t' are node-disjoint except $t[0] = t'[0]$ and their last nodes.*

2. *There is no edge $A \to z$ and $\mathit{DomMap}[z] = z$.*

Proof. By structural assumption of the DAG there is no other incoming edge into z, if $A \to z$. However, the path $A \to z$ is not edge-disjoint with itself. Therefore, we may assume that there is no direct edge from $A \to z$.

$\underline{(1) \implies (2)}$ In that case $t \in A \to^{+} q \to z$ and $t' \in A \to^{+} q' \to z$ with $q \neq q'$. We apply Theorem 2 and obtain that $\mathit{DomMap}[q] \neq \mathit{DomMap}[q']$. Further $\bot \notin \{\, \mathit{DomMap}[q], \mathit{DomMap}[q'] \,\}$ because of Lemma 5. Since z is no immediate successor of A, $\mathit{DomMap}[z] = \mathit{DomMap}[q] \overset{z}{\sqcup} \mathit{DomMap}[q'] \overset{z}{\sqcup} X$ for some $X \in \mathbb{V}'$. By applying Definition 25 for $\mathit{DomMap}[q] \neq \mathit{DomMap}[q']$, we get $\mathit{DomMap}[z] = z$

$\underline{(2) \implies (1)}$ Assume that $\mathit{DomMap}[z] = z$. Since z is no immediate successor of A, $\mathit{DomMap}[z] = \bigsqcup_{q \to z}^{z} \mathit{DomMap}[q]$. Hence, the only way that $\mathit{DomMap}[z] = z$ if z is that there are two predecessors of z, $q \to z$ and $q' \to z$ with $\mathit{DomMap}[q] \neq \mathit{DomMap}[q']$ and $\bot \notin \{\, \mathit{DomMap}[q], \mathit{DomMap}[q'] \,\}$. Application of Definition 25 and yields that that there are two almost node-disjoint paths $t \in A \to^{+} q$ and $t \in A \to^{+} q'$ such that one of them starts in a divergent edge out of A. We extend the path to $t_z = t \to z$ and $t'_z = t' \to z$. Clearly, t_z and t'_z are edge disjoint because t and t' are almost node-disjoint. Also $t_z, t'_z \in A \to^{+} z$ by construction.

□

Appendix B.

Properties of Partial Linearization

We prove three properties of partial linearization that simplify the design of the vectorizer pipeline. The first, preservation of dominance implies that partial linearization preserves all SSA properties. The second, preservation of uniform control dependence implies that if a block mask is uniform then the vector code generator does not have to predicate the block at all. Finally, preservation of uniform branches yields a simple criterion under which uniform branches are preserved. This, in turn, simplifies optimizations for divergent control such as BOSCC, as shown in Section 9.2.

B.1. Extended Notation

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.

B.1.1. 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.2. Preservation of Uniform Control Dependence

Lemma 2. *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 2 that was used as an unproven lemma in the proof of Theorem 4.1.

B.2.1. Auxiliary Lemmas

Lemma 7.

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

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 $(b, 0, c) \in \mathtt{E}_\ell$. Otherwise, if $x \neq c$, then $(b, 0, x) \in \mathtt{E}_\ell$ 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 $(b, 0, s) \in \mathtt{E}_\ell$ 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 $(b, i, s) \in \mathtt{E}$: If $next@b \neq c$ then $(b, i, next@b) \in \mathtt{E}_\ell$ and $(next@b, c) \in D@b$ since $c \in T@b$ and $c \neq next@b$. Otherwise, if $next@b = c$ then $(b, i, next@b) \in \mathtt{E}_\ell$. □

Lemma 8. $c \in T@b \implies c \succ_\ell^{PD} b$

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 7, 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 9. $a \succeq^{PD} x \implies a \succeq_\ell^{PD} x$

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 $\mathtt{E}$ it holds that $a \succeq^{PD} p$. By the induction hypothesis therefore $a \succeq_\ell^{PD} p$. For every edge $(x, i, next@x) \in \mathtt{E}_\ell$ 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 8 with $a \succeq_\ell^{PD} p$. Therefore, in general $a \succ_\ell^{PD} x$. □

Lemma 10. $\mathit{uni}(a) \implies T@a = \emptyset$

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

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

Outer induction step We may assume that given $\mathit{uni}(a)$ it holds that $\forall d \in \mathit{cdepB}(a).T@d = \emptyset$. This is because $\mathit{uni}(a)$ implies $\mathit{uni}(\mathit{cdep}(a))$. 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 $s \to p \in \mathtt{E}$ during which processing the pair $(\mathit{next}@s, p)$ 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 $s \to p \in \mathit{cdep}(a)$. With $\mathit{uni}(a)$ it follows that s has a uniform branch and the outer induction hypothesis implies that $T@s = \emptyset$. Therefore, always $(\mathit{next}@s, p) \notin D@s$ *after* line 10, for any such $s \to p \in \mathtt{E}$. This contradicts $p \in T@x$ for any $x \in V$.

So, if $t \in T@p$ due to $(\mathit{next}@q, t) \in D@q$ with $\mathit{next}@q = p$ then $q \to p \in \mathtt{E}$. However, then again $q \to p \in \mathit{cdep}(a)$ and q must have a uniform branch and the outer induction hypothesis yields $T@q = \emptyset$. Thus, $(\mathit{next}@q, t) \notin D@q$ *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 $\mathit{next}@s = p$ and $(p, t) \in D@s$ after the iteration.

We distinguish three cases for s:

Case 1. $s \to p \in \mathit{cdep}(a)$. 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 $\forall s \to n \in \mathtt{E}.a \succeq^{PD} n$. Therefore, regardless whether s has a uniform or varying branch it holds that $a \succeq^{PD} t$, which contradicts the assumption.

Case 3. $s \to p \notin cdep(a) \;\wedge\; a \not\succeq^{PD} s$. We know that $s \to p \notin \mathtt{E}$ because otherwise $s \to p$ would be a control dependence of a. Hence, there must be a different $q \in V$ with $q \to p \in \mathtt{E}$, 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 $q \to p \in cdep(a)$. 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 $s \to p \notin \mathtt{E}$, there must be in particular such a node q with $q \to p \in \mathtt{E}$ and a path $\pi' \in q \to^* x \to s$ 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 11. *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 10. 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 12. *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. $\exists i.(p, i, b) \in \mathtt{E}$
If $k \succeq^{PD} t$ for $t \in T@p$ then by the induction hypothesis, $k \succeq^{PD} p$. Further, since $p \to b \in \mathtt{E}$, immediately $k \succeq^{PD} b$.

Case 2. $\nexists i.(p, i, b) \in \mathtt{E}$
In this case $b = \mathit{next}@p \in T@p$. By the induction hypothesis with $t \in T@p$, $k \succeq^{PD} p$. So, it follows from Lemma 11 with $\mathit{uni}(k)$ that $\forall t \in T@p.k \succeq^{PD} t$ and in particular $k \succeq^{PD} \mathit{next}@p = b$.

□

Lemma 13. *if* $\forall a \to b \in \mathtt{E}.\mathit{uni}(a \to b)$ *then*
$\forall b.a \to b \in \mathtt{E} \iff a \to b \in \mathtt{E}_\ell$

Proof. $\mathit{uni}(a \to b)$ implies that a has a uniform branch and thus $\forall a \to b \in E.\mathit{uni}(a \to b)$. Since $\mathit{uni}(a)$ then $T@a = \emptyset$ by Lemma 10. Because of that $a \to b \in \mathtt{E}$ implies $a \to b \in \mathtt{E}_\ell$ by the algorithm. This means that $|\{\, b \mid a \to b \in \mathtt{E}_\ell \,\}| \geq |\{\, b \mid a \to b \in \mathtt{E}\} \,\}|$. However, the algorithm will only reduce the degree of branches. This means that $|\{\, b \mid a \to b \in \mathtt{E}_\ell \,\}| \leq |\{\, b \mid a \to b \in \mathtt{E}\} \,\}|$. Thus, $\forall b.(a \to b \in \mathtt{E} \iff a \to b \in \mathtt{E}_\ell)$. □

Lemma 14. *if* $\mathit{uni}(a)$ *then* $[a \succeq^{PD} b \impliedby a \succeq^{PD}_\ell b]$

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:

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

In the following assume $\mathit{uni}(a)$. 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 9 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 $(b, i, \mathit{next}@b) \in \mathtt{E}_\ell$ with $\mathit{next}@b \in T@b$.
In this case, it follows directly from Lemma 12 that $a \not\succeq^{PD} b$ implies $a \not\succeq^{PD} \mathit{next}@b$. By the induction hypothesis for $\mathit{next}@b$, we conclude that $a \not\succeq^{PD}_\ell \mathit{next}@b$. Since $b \to \mathit{next}@b \in \mathtt{E}_\ell$ therefore also $a \not\succeq^{PD}_\ell b$.

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

Sub case 2.1. b has a divergent branch.
Assume there was a $b \to s \in \mathtt{E}$ with $a \not\succeq^{PD} b$ and $a \succeq^{PD} s$. This implies that $b \to s \in \mathit{cdep}(a)$. However, as $\mathit{uni}(a)$ 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 $\forall b \to s \in \mathtt{E}.a \not\succeq^{PD} s$.

So, if $b \to next@b \in \mathtt{E}$ then $a \not\succeq^{PD} next@b$. We apply the induction hypothesis to obtain $a \not\succeq_\ell^{PD} next@b$ and finally $a \not\succeq_\ell^{PD} b$.

Sub case 2.2. b has a uniform branch.
Since $a \not\succeq^{PD} b$ there must be an edge $b \to s \in \mathtt{E}$ 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 $a \not\succeq_\ell^{PD} s$. Therefore, also $a \not\succeq_\ell^{PD} b$. □

B.2.2. Main Proof

This is the main proof of Lemma 2.

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 $c \succeq_\ell^{PD} b$ and $c \not\succeq_\ell^{PD} a$ and $a \to b \in \mathtt{E}_\ell$. By Lemma 9 and Lemma 14, given that $uni(c)$, it follows that $c \succeq^{PD} b$ and $c \not\succeq^{PD} a$. It remains to show that $a \to b \in \mathtt{E}$. Assume this was not the case, that is $a \to b \in \mathtt{E}_\ell$ and $a \to b \notin \mathtt{E}$. As $a \to b \in \mathtt{E}_\ell$, we get $b \in T@a$ and therefore, by Lemma 8, $b \succeq_\ell^{PD} a$. Since also $c \succeq_\ell^{PD} b$ this contradicts the assumption that $c \not\succeq_\ell^{PD} a$. Thus, $a \to b \in \mathtt{E}$.

Finally, from $a \to b \in \mathtt{E}$ 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 13, $a \to b \in \mathtt{E}_\ell$ because a has a uniform branch and $a \to b \in \mathtt{E}$. $a \to b \in cdep(c)$ also implies $c \succeq^{PD} b$ and $c \not\succeq^{PD} a$ by definition of control dependence. However, by Lemma 9, $c \succeq^{PD} b$ implies $c \succeq_\ell^{PD} b$ and since $uni(c)$ it also follows by Lemma 14 that $c \not\succeq^{PD} a$ implies $c \not\succeq_\ell^{PD} a$. In short, $a \to b \in \mathtt{E}_\ell$ and $c \succeq_\ell^{PD} b$ and $c \not\succeq_\ell^{PD} a$ and so by definition $a \to b \in cdep_\ell(c)$. □

B.3. Preservation of Uniform Branches

Theorem 5. *Given a dominance-compact block index, partial linearization will preserve an edge $b \to y \in \mathtt{E}$ if there exists a block $d \in V$ with the following properties in G:*

1. $d \succeq^D b \land d \succ^D y$ *(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 5. We will prove that the edges that $d \in V$ dominates in the partially linearized subgraph G_ℓ^d 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 26.

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 $e \in \mathtt{E} \setminus \mathtt{E}^d$ such that both instances can execute in lock step over all of $b \in V$. Note that the two instances operate on the same block index, that is $\mathit{BlockIndex}(b) = \mathit{BlockIndex}^d(b)$ for $b \in V^d$.

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

Definition 26. *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 $\mathtt{E}_\ell^d@b \sim \mathtt{E}_\ell@b$ where these are defined as:*

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

$$\forall x, d \succ^D y.\ \Big[(x,y) \in D^d@b \iff (x,y) \in D@b\Big]$$

$\mathtt{E}_\ell^d@b \sim \mathtt{E}_\ell@b$ *iff*

$$\forall d \succeq^D x \ \land\ d \succ^D y.\ \Big[x \to y \in \mathtt{E}_\ell^d@b \iff x \to y \in \mathtt{E}_\ell@b\Big]$$

B.3.1. Main Proof

Theorem 8. *Partial linearization maintains the equivalence relation of Definition 26.*

Proof. We will prove this by induction over the two instances of the algorithm. The induction hypothesis states that the equivalence relation of Definition 26 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 $\mathtt{E}_\ell^d = \mathtt{E}_\ell = \emptyset$.

Induction step (case $d \not\succ^D$ *next@b*) $\underline{D^d@b \sim D@b}$ Assume there was a $(\mathit{next}@b, y) \in D@b$ with $d \succ^D y$ *after* the outer loop iteration for b. Then $\mathit{next}@b < y$ and further $\mathit{next}@b < d$ because the block index is dominance compact. There must be an edge $p \to y \in \mathtt{E}$ with $p \leq b < \mathit{next}@b \leq d < y$ 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.

$\underline{\mathtt{E}_\ell^d@b \sim \mathtt{E}_\ell@b}$: The $\forall$-quantifier in the definition of $\mathtt{E}_\ell^d@b \sim \mathtt{E}_\ell@b$ does not quantify over edges $b \to \mathit{next}@b \in \mathtt{E}_\ell@b$ with $d \not\succeq^D \mathit{next}@b$. These are the only kind of edges added to $\mathtt{E}_\ell$ and $\mathtt{E}_\ell^d$ 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 $p \to next@b \in \mathtt{E}$ 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 $p \to next@b \in \mathtt{E}$.

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 (Definition 22). Further, $D \sim D^d$ and $\mathtt{E}_\ell \sim \mathtt{E}^d_\ell$ *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 $\mathtt{E}_\ell@b \sim \mathtt{E}^d_\ell@b$ *after* the step.

Case 1. Inner loop step for uniform branch in b.
Let $(b, i, s) \in \mathtt{E}$ 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, $next@b = next^d@b$.

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

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 $D \leftarrow D \setminus \{\, (b, s) \mid (b, s) \in D \,\}$ 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 $b \in V$. □

Glossary

activation The bit mask that determines, which threads will actively execute a scheduled basic block in a P-LLVM program. The activation is defined as the conjunction of the block predicate and the control mask (see Definition 8). 38, 69, 178, 179

block index A block index is a topological enumeration of all blocks that is compact in all loops and dominated block sets. An enumeration is compact in any given finite set, if the set is empty or the difference between the maximum and minimum number assigned to the blocks in the set plus one is equal to the size of the set (see Section 2.5). 8, 101, 102

BOSCC gadget Branch on Superword Condition Code (BOSCC) is a branch that skips a part of the CFG, which would otherwise execute with an all-false activation (see Section 9.2). 111

brush projection The brush projection describes how the coordinates in a d-dimensional tensor are mapped to one-dimensional SIMD registers (see Section 11.5). 132, 136, 138

CFG Control-Flow Graph (see Section 2.2). 7, 20, 79, 104, 109, 142, 144, 146, 178–180

concurrent next block Two blocks are *concurrent next blocks* when at any point in the execution of a P-LLVM program there is one thread with an enabled control mask for each of the blocks (see Definition 20). If this is the case, the threads have the potential to reconverge later in the execution, which may cause ϕ nodes to become varying because each thread selects a different incoming value. 83

control mask The control mask of a thread is true if the scheduled block is compatible with the thread's control state (see Definition 6). The conjunction of the control mask and a block's predicate form the activation, which determines whether a thread executes a scheduled block actively. 53, 58, 74, 156, 178, 180

control state Every thread that executes a P-LLVM program holds in its execution state two block labels: The *next block* (ℓ_{next}) is the block that the thread has branched to but has not yet executed. The *incoming block* (ℓ_{in}) is the predecessor of the next block from which the thread has branched to the next block. The pair of incoming and next block is called the *control state* of the thread. The next block is a specific block label, $\Box(r \in \mathcal{V})$ to show that the thread has executed a return statement with return value r, or $\top$ to signal that any successor of the incoming block is a valid branch target for this thread. The incoming block is a specific block label, or $\bot$ in the initial state where the entry block is the next block. 178

control uniform A terminator is control-uniform if all threads that actively execute it advance to the same successor. A P-LLVM program is control-uniform if all its terminators are control-uniform. 15, 16, 19, 31, 34, 66, 100, 179, 180

control-divergence analysis When threads part ways at a divergent branch and reach the same ϕ node from different incoming blocks, the divergent branch has induced divergence in the ϕ node. The Control-Divergence Analysis finds for a given branch in a CFG all basic blocks whose ϕ nodes become divergent if the branch is divergent (see Chapter 7). The divergence analysis queries the control-divergence analysis to detect divergent ϕ nodes after it has identified a divergent branch. 2, 21, 22, 58, 81, 83, 144, 179

DAG A Directed Acyclic Graph (DAG) is a CFG without cycles (see Section 2.2). 81, 99, 180

definition mask Due to thread activation, a thread in a P-LLVM program may execute an instruction that syntactically defines a SSA variable without defining it semantically. The definition mask is true, only if the execution of an SSA variable assignment is considered a defining assignment. The definition mask differs from the activation for the current block for ϕ nodes and instructions with the **total** modifier (see Definition 10). 53, 68

divergence analysis The analysis that identifies non-uniform branches and variables in a program (see Chapter 6). 2, 7, 14–16, 20–22, 31, 53, 58, 66, 69, 79–82, 118–121, 123, 129, 144, 155, 162, 179

divergence lattice The divergence analysis assigns elements of the divergence lattice to program variables and branches in P-LLVM programs. The elements represent what relation exists between the values of the variable for different threads as they execute the defining instruction in lock step. A divergence lattice always contains a uniform element to identify variables that hold the same value across all threads and branches that are control uniform. In case of RV, the elements of the divergence lattice are called vector shapes. In case of TensorRV, the elements are called tensor shapes. 67, 180

divergent loop transform The transformation that makes loops control uniform (see Section 9.1). 28, 31, 46, 66, 100, 111–114, 117, 149

greedy schedule The execution of a P-LLVM program is driven by a runtime scheduler, which produces a potentially infinite sequence of block labels that all thread execute in lock step. Among all possible schedules for a program, only a subset materializes in the SIMD code that RV generates. This is the class of *Greedy Schedules* (see Definition 12). 58, 62, 64, 80, 83, 88, 145

LCSSA In Loop-Closed SSA form, the only allowed users of a loop-defined variable outside the defining loop are ϕ nodes in immediate exit blocks of the loop (see Section 2.4). Compilers such as GCC and LLVM ship with transformations that establish LCSSA form. In the course of the transformation, new single-entry ϕ nodes are inserted in immediate loop exit blocks to break data-flow edges that violate the LCSSA property. 69

lock step Lock-step execution is a form of data-parallel execution where a thread array executes the same program on different inputs. In lock-step execution, the only way to execute an instruction is for all threads of the thread array to execute it concurrently. 25, 27–29, 38, 179

masked concretization The vector shapes of non-total instructions and non-shadow ϕ nodes are only meaningful for those threads that execute the operation actively. The masked concretization has an extra activation vector input to mask out those lanes in the concretization, which in effect allows any value on masked-off lanes (see Section 6.2). 79

partial control-flow linearization The partial if-conversion technique discussed in Chapter 8 Partial linearization takes a DAG with uniform and divergent branches and makes it control uniform. It retains the uniform branches in the process. 28, 66

progress Constraints the space of legal schedules: at least one thread must have an active control mask and branches on uniform conditions must not cause control divergence (see Definition 14). 59

reducible CFG A CFG where every depth-first search yields the same set of retreating edges and for every retreating edge $x \to y \in \mathtt{E}$ the sink y dominates the source x (see Section 2.4). 7, 84, 100

tensor brush In loop nest tensorization multiple nested loop are vectorized at once. There is not one vectorization width but instead one width per loop. The tuple of those widths is the tensor brush (see Definition 24). 131, 132, 136, 160

tensor shape An element of a multi-dimensional, abstract divergence lattice (see Section 11.3). Tensor shapes are the elements of the divergence lattice for TensorRV. 132, 179

thread array The array of threads execution the same P-LLVM program (see Definition 2). 17, 32, 39, 44, 66, 119, 132, 180

uniform A uniform variable always holds the same value for all threads that execute the defining instruction in lock-step. A uniform branch goes the same way for all threads that execute it in lock-step - the branch is control uniform. Branches and variables that are non-uniform are called divergent or varying. 100, 179, 180

vector shape An element of a single-dimensional, abstract divergence lattice. Vector shapes are the elements of the divergence lattice for RV. We define minimal requirements for vector shape lattices in Section 6.1, e.g. the existence of a uniform vector shape. Chapter 10 presents a more expressive set of vector shapes that RV exploits to generate faster SIMD code. 67, 70, 71, 73, 118, 119, 121, 122, 124, 126, 128, 179

Bibliography

Sx-aurora tsubasa architecture guide revision 1.1. https://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf, 2018. Accessed: 2019.09.13.

User guide for amdgpu backend. http://llvm.org/docs/AMDGPUUsage.html, 2019. Accessed: 2019.12.06.

[rfc] redefine ‘convergent‘ in terms of dynamic instances. https://reviews.llvm.org/D68994, 2019. Accessed: 2019.12.06.

M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 147–160, 1999. doi: 10.1145/292540.292555. URL https://doi.org/10.1145/292540.292555.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN 0-201-10088-6. URL http://www.worldcat.org/oclc/12285707.

A. Aiken and D. Gay. Barrier inference. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 342–354, 1998. doi: 10.1145/268946.268974. URL https://doi.org/10.1145/268946.268974.

T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009, New Orleans, Louisiana, USA, August 1-3, 2009*, pages 145–149, 2009. doi: 10.2312/EGGH/HPG09/145-150. URL https://doi.org/10.2312/EGGH/HPG09/145-150.

E. Albert, K. Knobe, J. D. Lukas, and G. L. S. Jr. Compiling fortran 8x array features for the connection machine computer system. In *Proceedings of the ACM/SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages and Systems, New Haven, Connecticut, USA, July 19-21, 1988*, pages 42–56, 1988. doi: 10.1145/62115.62121. URL http://doi.acm.org/10.1145/62115.62121.

J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*, pages 177–189, 1983. doi: 10.1145/567067.567085. URL http://doi.acm.org/10.1145/567067.567085.

R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, 1987. doi: 10.1145/29873.29875. URL `http://doi.acm.org/10.1145/29873.29875`.

K. A. Alon Amid et al. RISC-V "V" Vector Extension. `https://github.com/riscv/riscv-v-spec/releases/download/0.7.1/riscv-v-spec-0.7.1.pdf`, 2019. Accessed: 2019.08.14.

B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 1–11, 1988. doi: 10.1145/73560.73561. URL `https://doi.org/10.1145/73560.73561`.

R. Alur, J. Devietti, O. S. N. Leija, and N. Singhania. Gpudrano: Detecting uncoalesced accesses in GPU programs. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 507–525, 2017. doi: 10.1007/978-3-319-63387-9_25. URL `https://doi.org/10.1007/978-3-319-63387-9_25`.

AMD. "vega" instruction set architecture. `https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf`, 2017. Accessed: 2019.09.13.

AMD. "rdna 1.0" instruction set architecture. `https://gpuopen.com/wp-content/uploads/2019/08/RDNA_Shader_ISA_5August2019.pdf`, 2019. Accessed: 2019.09.13.

J. Anantpur and R. Govindarajan. Taming control divergence in gpus through control flow linearization. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 133–153, 2014. doi: 10.1007/978-3-642-54807-9_8. URL `https://doi.org/10.1007/978-3-642-54807-9_8`.

A. Anderson, A. Malik, and D. Gregg. Automatic vectorization of interleaved data revisited. *TACO*, 12(4):50:1–50:25, 2016. doi: 10.1145/2838735. URL `https://doi.org/10.1145/2838735`.

A. W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, 1998. doi: 10.1145/278283.278285. URL `https://doi.org/10.1145/278283.278285`.

ARM. Introducing the new Armv8.1-M architecture. URL `https://pages.arm.com/introduction-armv8.1m.html`.

Arm. Arm a64 instruction set architecture - armv8-a architecture profile. `https://static.docs.arm.com/ddi0596/d/ISA_A64_xml_v85A-2019-06_OPT.pdf`, 2019. Accessed: 2019.09.13.

D. I. August, W. W. Hwu, and S. A. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, 27(5):381–423, 1999. doi: 10.1023/A:1018787007582. URL `https://doi.org/10.1023/A:1018787007582`.

J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*, pages 149–159, 1996.

doi: 10.1145/231379.231409. URL `https://doi.org/10.1145/231379.231409`.

S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. Flexvec: auto-vectorization for irregular loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 697–710, 2016. doi: 10.1145/2908080.2908111. URL `http://doi.acm.org/10.1145/2908080.2908111`.

H. Bahmann, N. Reissmann, M. Jahre, and J. C. Meyer. Perfect reconstructability of control flow from demand dependence graphs. *TACO*, 11(4):66:1–66:25, 2014. doi: 10.1145/2693261. URL `http://doi.acm.org/10.1145/2693261`.

R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control, data, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 257–271, 1990. doi: 10.1145/93542.93578. URL `https://doi.org/10.1145/93542.93578`.

J. E. Barnes. A modified tree code: don't laugh; it runs. *Journal of Computational Physics*, 87(1): 161–170, 1990.

W. Baxter and H. R. B. III. The program dependence graph and vectorization. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 1–11, 1989. doi: 10.1145/75277.75278. URL `https://doi.org/10.1145/75277.75278`.

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

D. Caballero, S. Royuela, R. Ferrer, A. Duran, and X. Martorell. Optimizing overlapped memory accesses in user-directed vectorization. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 393–404, 2015. doi: 10.1145/2751205.2751224. URL `https://doi.org/10.1145/2751205.2751224`.

L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Newport Beach, California, USA, October 12-16, 1999*, pages 245–255, 1999. doi: 10.1109/PACT.1999.807561. URL `https://doi.org/10.1109/PACT.1999.807561`.

S. Chakraborty and V. Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 100–110, 2017. URL `http://dl.acm.org/citation.cfm?id=3049844`.

A. Chandrasekhar, G. Chen, P. Chen, W. Chen, J. Gu, P. Guo, S. H. P. Kumar, G. Lueh, P. Mistry, W. Pan, T. Raoux, and K. Trifunovic. IGC: the open source intel graphics compiler. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 254–265, 2019. doi: 10.1109/CGO.2019.8661189. URL `https:`

//doi.org/10.1109/CGO.2019.8661189.

S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797. URL https://doi.org/10.1109/IISWC.2009.5306797.

W. Chuang, B. Calder, and J. Ferrante. Phi-predication for light-weight if-conversion. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*, pages 179–192, 2003. doi: 10.1109/CGO.2003.1191544. URL https://doi.org/10.1109/CGO.2003.1191544.

N. Clark, A. Hormati, S. Yehia, S. A. Mahlke, and K. Flautner. Liquid SIMD: abstracting SIMD hardware using lightweight dynamic mapping. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*, pages 216–227, 2007. doi: 10.1109/HPCA.2007.346199. URL https://doi.org/10.1109/HPCA.2007.346199.

S. Collange. Identifying scalar behavior in cuda kernels. Technical report, ENS Lyon, 2011a. URL https://hal.archives-ouvertes.fr/hal-00555134/.

S. Collange. Stack-less SIMT reconvergence at low cost. (hal-00622654), 2011b.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971. doi: 10.1145/800157.805047. URL https://doi.org/10.1145/800157.805047.

B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*, pages 320–329, 2011. doi: 10.1109/PACT.2011.63. URL https://doi.org/10.1109/PACT.2011.63.

R. Cytron and R. Gershbein. Efficient accomodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 36–45, 1993. doi: 10.1145/155090.155094. URL https://doi.org/10.1145/155090.155094.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4): 451–490, 1991. doi: 10.1145/115372.115320. URL http://doi.acm.org/10.1145/115372.115320.

G. F. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD re-convergence at thread frontiers. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 477–488, 2011. doi: 10.1145/2155620.2155676. URL https://doi.org/10.1145/2155620.2155676.

A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 82–93, 2004. doi:

10.1145/996841.996853. URL https://doi.org/10.1145/996841.996853.

A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt. A scalable multi-path microarchitecture for efficient GPU control flow. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pages 248–259, 2014. doi: 10.1109/HPCA.2014.6835936. URL https://doi.org/10.1109/HPCA.2014.6835936.

P. Estérie, M. Gaunard, J. Falcou, J. Lapresté, and B. Rozoy. Boost.simd: generic programming for portable simdization. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 431–432, 2012. doi: 10.1145/2370816.2370881. URL https://doi.org/10.1145/2370816.2370881.

C. A. Farrell and D. H. Kieronska. Formal specification of parallel SIMD execution. *Theor. Comput. Sci.*, 169(1):39–65, 1996. doi: 10.1016/S0304-3975(96)00113-2. URL https://doi.org/10.1016/S0304-3975(96)00113-2.

P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991. doi: 10.1007/BF01407931. URL https://doi.org/10.1007/BF01407931.

P. Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992a. doi: 10.1007/BF01407835. URL https://doi.org/10.1007/BF01407835.

P. Feautrier. Some efficient solutions to the affine scheduling problem. part II. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992b. doi: 10.1007/BF01379404. URL https://doi.org/10.1007/BF01379404.

J. Ferrante and M. Mace. On linearizing parallel code. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 179–189, 1985. doi: 10.1145/318593.318636. URL http://doi.acm.org/10.1145/318593.318636.

J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. doi: 10.1145/24039.24041. URL http://doi.acm.org/10.1145/24039.24041.

M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9): 948–960, 1972. doi: 10.1109/TC.1972.5009071. URL https://doi.org/10.1109/TC.1972.5009071.

A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 93:110, 2011.

M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93 (2):216–231, 2005. doi: 10.1109/JPROC.2004.840301. URL https://doi.org/10.1109/JPROC.2004.840301.

S. Fuller. Motorola's altivec tm technology. https://www.nxp.com/assets/documents/data/en/fact-sheets/ALTIVECWP.pdf, 1998. Accessed: 2019.08.14.

W. W. L. Fung, I. Sham, G. L. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*, pages 407–420, 2007. doi: 10.1109/MICRO.2007.30. URL https://doi.org/10.1109/MICRO.2007.30.

I. Georgiev and P. Slusallek. Rtfact: Generic concepts for flexible and high performance ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 115–122. IEEE, 2008.

J. Glossner, P. Blinzer, and J. Takala. Hsa-enabled dsps and accelerators. In *2015 IEEE Global Conference on Signal and Information Processing, GlobalSIP 2015, Orlando, FL, USA, December 14-16, 2015*, pages 1407–1411, 2015. doi: 10.1109/GlobalSIP.2015.7418430. URL https://doi.org/10.1109/GlobalSIP.2015.7418430.

A. G. Gray and A. W. Moore. N-body'problems in statistical learning. In *Advances in neural information processing systems*, pages 521–527, 2001.

T. K. V. W. Group. Vulkan 1.1 - a specification. https://www.khronos.org/registry/vulkan/specs/1.1/pdf/vkspec.pdf, 2019. Accessed: 2019.09.13.

M. D. Guzzi, D. A. Padua, J. Hoeflinger, and D. H. Lawrie. Cedar fortran and other vector and parallel fortran dialects. *The Journal of Supercomputing*, 4(1):37–62, 1990. doi: 10.1007/BF00162342. URL https://doi.org/10.1007/BF00162342.

A. Habermaier and A. Knapp. On the correctness of the SIMT execution model of gpus. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 316–335, 2012. doi: 10.1007/978-3-642-28869-2_16. URL https://doi.org/10.1007/978-3-642-28869-2_16.

M. Haidl, S. Moll, L. Klein, H. Sun, S. Hack, and S. Gorlatch. 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@SC 2017, Denver, CO, USA, November 13, 2017*, pages 7:1–7:12, 2017. doi: 10.1145/3148173.3148185. URL http://doi.acm.org/10.1145/3148173.3148185.

C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009. doi: 10.1007/s10207-009-0086-1. URL https://doi.org/10.1007/s10207-009-0086-1.

P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1990, Dallas, TX, USA, August 6-10, 1990*, pages 289–298, 1990. doi: 10.1145/97879.97911. URL https://doi.org/10.1145/97879.97911.

P. Havlak. Construction of thinned gated single-assignment form. In *Languages and Compilers for Parallel Computing, 6th International Workshop, Portland, Oregon, USA, August 12-14, 1993,* *Proceedings*, pages 477–499, 1993. doi: 10.1007/3-540-57659-2_28. URL https://doi.org/10.1007/3-540-57659-2_28.

M. S. Hecht. *Flow analysis of computer programs.* Elsevier Science Inc., 1977.

M. S. Hecht and J. D. Ullman. Flow graph reducibility. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 238–250, 1972. doi: 10.1145/800152.804919. URL http://doi.acm.org/10.1145/800152.804919.

M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974. doi: 10.1145/321832.321835. URL http://doi.acm.org/10.1145/321832.321835.

N. Hegde, J. Liu, and M. Kulkarni. Treelogy: a benchmark suite for tree traversal applications. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*, pages 227–228, 2016. doi: 10.1109/IISWC.2016.7581286. URL https://doi.org/10.1109/IISWC.2016.7581286.

A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 981–991, 2016. doi: 10.1109/SC.2016.83. URL https://doi.org/10.1109/SC.2016.83.

R. A. Hendrickson. Array processing extensions to FORTRAN. In *Proceedings of the 1979 Annual Conference, Detroit, Michigan, USA, October 29-31, 1979.*, pages 175–178, 1979. doi: 10.1145/800177.810061. URL http://doi.acm.org/10.1145/800177.810061.

T. Henretty, K. Stock, L. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector SIMD architectures. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 225–245, 2011. doi: 10.1007/978-3-642-19861-8_13. URL https://doi.org/10.1007/978-3-642-19861-8_13.

L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. *Theor. Comput. Sci.*, 248(1-2):3–27, 2000. doi: 10.1016/S0304-3975(00)00048-7. URL https://doi.org/10.1016/S0304-3975(00)00048-7.

K. Hou, H. Wang, and W. Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015*, pages 383–392, 2015. doi: 10.1145/2751205.2751247. URL https://doi.org/10.1145/2751205.2751247.

Intel. Intel® architecture instruction set extensions programming reference (pdf). https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference. Accessed: 2019.08.09.

ISO 19570:2018. Technical specification for c++ extensions for parallelism. Standard, International Organization for Standardization, Geneva, CH, Nov. 2018.

Y. Jo, M. Goldfarb, and M. Kulkarni. Automatic vectorization of tree traversals. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, September 7-11, 2013*, pages 363–374, 2013. doi: 10.1109/PACT.2013.6618832. URL `https://doi.org/10.1109/PACT.2013.6618832`.

N. Johnson and A. Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 1–16, 2003. doi: 10.1007/3-540-36579-6_1. URL `https://doi.org/10.1007/3-540-36579-6_1`.

N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977. doi: 10.1007/BF00290339. URL `https://doi.org/10.1007/BF00290339`.

R. Karrenberg. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs*. Springer, 2015. ISBN 978-3-658-10112-1. doi: 10.1007/978-3-658-10113-8. URL `https://doi.org/10.1007/978-3-658-10113-8`.

R. Karrenberg and S. Hack. Whole-function vectorization. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pages 141–150, 2011. doi: 10.1109/CGO.2011.5764682. URL `https://doi.org/10.1109/CGO.2011.5764682`.

R. Karrenberg and S. Hack. Improving performance of opencl on cpus. In *Compiler Construction - 21st International Conference, CC 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 1–20, 2012. doi: 10.1007/978-3-642-28652-0_1. URL `https://doi.org/10.1007/978-3-642-28652-0_1`.

R. Karrenberg, M. Kosta, and T. Sturm. Presburger arithmetic in memory access optimization for data-parallel languages. In *Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings*, pages 56–70, 2013. doi: 10.1007/978-3-642-40885-4_5. URL `https://doi.org/10.1007/978-3-642-40885-4_5`.

A. Kerr, G. F. Diamos, and S. Yalamanchili. Dynamic compilation of data-parallel kernels for vector processors. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pages 23–32, 2012. doi: 10.1145/2259016.2259020. URL `http://doi.acm.org/10.1145/2259016.2259020`.

J. Kessenich, B. Ouriel, and R. Krisch. Spir-v specification, 2018.

J. Kessenich, D. Baldwin, and R. Rost. The opengl shading language, version 4.60.7. `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf`, 2019. Accessed: 2019.09.13.

D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. M. Chapman. LLVM parallel intermediate representation: design and evaluation using openshmem communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, pages 2:1–2:8, 2015. doi: 10.1145/2833157.2833158. URL `https://doi.org/10.1145/2833157.2833158`.

M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell. Extending openmp* with vector constructs for modern multicore SIMD architectures. In *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*, pages 59–72, 2012. doi: 10.1007/978-3-642-30961-8_5. URL `https://doi.org/10.1007/978-3-642-30961-8_5`.

K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi. Performance evaluation of a vector supercomputer sx-aurora TSUBASA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 54:1–54:12, 2018. URL `http://dl.acm.org/citation.cfm?id=3291728`.

M. Kong, R. Veras, K. Stock, F. Franchetti, L. Pouchet, and P. Sadayappan. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 127–138, 2013. doi: 10.1145/2462156.2462187. URL `http://doi.acm.org/10.1145/2462156.2462187`.

M. Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, Goethe University Frankfurt am Main, 2015. URL `http://publikationen.ub.uni-frankfurt.de/frontdoor/index/index/docId/38415`.

O. Krzikalla, F. Wende, and M. Höhnerbach. Dynamic SIMD vector lane scheduling. In *High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers*, pages 354–365, 2016. doi: 10.1007/978-3-319-46079-6_25. URL `https://doi.org/10.1007/978-3-319-46079-6_25`.

M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 65–76, 2009. doi: 10.1109/ISPASS.2009.4919639. URL `https://doi.org/10.1109/ISPASS.2009.4919639`.

H. Lang, A. Kipf, L. Passing, P. A. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, pages 5:1–5:8, 2018. doi: 10.1145/3211922.3211928. URL `https://doi.org/10.1145/3211922.3211928`.

S. Larsen and S. P. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 145–156,

2000. doi: 10.1145/349299.349320. URL https://doi.org/10.1145/349299.349320.

C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

M. Lattuada and F. Ferrandi. Exploiting vectorization in high level synthesis of nested irregular loops. *Journal of Systems Architecture - Embedded Systems Design*, 75:1–14, 2017. doi: 10.1016/j.sysarc.2017.03.001. URL https://doi.org/10.1016/j.sysarc.2017.03.001.

Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 32:1–32:11, 2013. doi: 10.1109/CGO.2013.6494995. URL https://doi.org/10.1109/CGO.2013.6494995.

Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanovic. Exploring the design space of SPMD divergence management on data-parallel architectures. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*, pages 101–113, 2014. doi: 10.1109/MICRO.2014.48. URL https://doi.org/10.1109/MICRO.2014.48.

R. Leißa. *Language Support for Programming High-Performance Code*. PhD thesis, Saarland University, Saarbr&quot;ucken, Germany, 2017. URL https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27016.

R. Leißa, I. Haffner, and S. Hack. Sierra: a SIMD extension for C++. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*, pages 17–24, 2014. doi: 10.1145/2568058.2568062. URL https://doi.org/10.1145/2568058.2568062.

R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt. Anydsl: a partial evaluation framework for programming high-performance libraries. *PACMPL*, 2 (OOPSLA):119:1–119:30, 2018. doi: 10.1145/3276489. URL https://doi.org/10.1145/3276489.

A. Levinthal and T. K. Porter. Chap - a SIMD graphics processor. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1984, Minneapolis, Minnesota, USA, July 23-27, 1984*, pages 77–82, 1984. doi: 10.1145/800031.808581. URL https://doi.org/10.1145/800031.808581.

Y. Liang, M. T. Satria, K. Rupnow, and D. Chen. An accurate GPU performance model for effective control flow divergence optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 35(7): 1165–1178, 2016. doi: 10.1109/TCAD.2015.2501303. URL https://doi.org/10.1109/TCAD.2015.2501303.

E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008. doi: 10.1109/MM.2008.31. URL https://doi.org/10.1109/MM.2008.31.

T. Lloyd, K. Ali, and J. N. Amaral. Gpucheck: Detecting cuda thread divergence with static analysis. 2019.

S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Oregon, USA, November 1992*, pages 45–54, 1992. doi: 10.1109/MICRO.1992.696999. URL `https://doi.org/10.1109/MICRO.1992.696999`.

M. Masten, E. Tyurin, K. Mitropoulou, E. Garcia, and H. Saito. Function/kernel vectorization via loop vectorizer. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 39–48, Nov 2018. doi: 10.1109/LLVM-HPC.2018.8639483.

D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic SIMD vectorization of fast fourier transforms for the larrabee and AVX instruction sets. In *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, pages 265–274, 2011. doi: 10.1145/1995896.1995938. URL `https://doi.org/10.1145/1995896.1995938`.

C. Mendis and S. P. Amarasinghe. goslp: globally optimized superword level parallelism framework. *PACMPL*, 2(OOPSLA):110:1–110:28, 2018. doi: 10.1145/3276480. URL `https://doi.org/10.1145/3276480`.

C. Mendis, A. Jain, P. Jain, and S. P. Amarasinghe. Revec: program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, pages 29–41, 2019. doi: 10.1145/3302516.3307357. URL `https://doi.org/10.1145/3302516.3307357`.

Microsoft. Waveactiveballot function. `https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/waveballot`, 2018. Accessed: 2019.08.03.

S. Moll and S. Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 543–556, 2018. doi: 10.1145/3192366.3192413. URL `http://doi.acm.org/10.1145/3192366.3192413`.

S. Moll, J. Doerfert, and S. Hack. Input space splitting for opencl. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 251–260, 2016. doi: 10.1145/2892208.2892217. URL `http://doi.acm.org/10.1145/2892208.2892217`.

S. Moll, S. Sharma, M. Kurtenacker, and S. Hack. Multi-dimensional vectorization in llvm. In *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'19, pages 3:1–3:8, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6291-7. doi: 10.1145/3303117.3306172. URL `http://doi.acm.org/10.1145/3303117.3306172`.

R. E. A. Moreira, S. Collange, and F. M. Q. Pereira. Function call re-vectorization. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 313–326, 2017. URL `http://dl.acm.org/citation.cfm?id=3018751`.

A. Munshi. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

C. J. Newburn, B. So, Z. Liu, M. D. McCool, A. M. Ghuloum, S. D. Toit, Z. Wang, Z. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pages 224–235, 2011. doi: 10.1109/CGO.2011.5764690. URL `https://doi.org/10.1109/CGO.2011.5764690`.

V. N. Ngo. *Parallel Loop Transformation Techniques for Vector-based Multiprocessor Systems.* PhD thesis, Minneapolis, MN, USA, 1995. UMI Order No. GAX94-33091.

J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, 2008. doi: 10.1145/1365490.1365500. URL `https://doi.org/10.1145/1365490.1365500`.

D. Novillo et al. Memory ssa-a unified approach for sparsely representing memory operations. In *Proc of the GCC Developers' Summit*. Citeseer, 2007.

D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *17th International Conference on Parallel Architecture and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, pages 2–11, 2008. doi: 10.1145/1454115.1454119. URL `http://doi.acm.org/10.1145/1454115.1454119`.

D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 132–143, 2006. doi: 10.1145/1133981.1133997. URL `http://doi.acm.org/10.1145/1133981.1133997`.

D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor SIMD: auto-vectorize once, run everywhere. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pages 151–160, 2011. doi: 10.1109/CGO.2011.5764683. URL `https://doi.org/10.1109/CGO.2011.5764683`.

NVIDIA. V100 GPU architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017. Accessed: 2019.11.20.

NVIDIA. Nvidia turing gpu architecture. `https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`, 2018. Accessed: 2019.08.03.

NVIDIA. PTX ISA version 6.4. `http://docs.nvidia.com/cuda/pdf/ptx_isa_6.4.pdf`, 2019a. Accessed: 2019.05.08.

NVIDIA. NVVM IR Specification 1.5. `https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html`, 2019b. Accessed: 2019.08.06.

A. OpenMP. OpenMp Application Programming Interface, Version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015. Accessed: 2019.08.08.

J. C. Park and M. Schlansker. *On predicated execution.* Hewlett-Packard Laboratories Palo Alto, California, 1991.

S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. P. Luebke, D. K. McAllister, M. McGuire, R. K. Morley, A. Robison, and M. Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, 2010. doi: 10.1145/1778765.1778803. URL https://doi.org/10.1145/1778765.1778803.

A. Pérard-Gayot, R. Membarth, P. Slusallek, S. Moll, R. Leißa, and S. Hack. A data layout transformation for vectorizing compilers. In *Proceedings of the 4th Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2018, Vienna, Austria, February 24, 2018*, pages 7:1–7:8, 2018. doi: 10.1145/3178433.3178440. URL http://doi.acm.org/10.1145/3178433.3178440.

A. Pérard-Gayot, R. Membarth, R. Leißa, S. Hack, and P. Slusallek. Rodent: generating renderers without writing a generator. *ACM Trans. Graph.*, 38(4):40:1–40:12, 2019. doi: 10.1145/3306346.3322955. URL https://doi.org/10.1145/3306346.3322955.

M. Pharr and W. R. Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012.

L. Plagne and K. Bojnourdi. Portable vectorization and parallelization of C++ multi-dimensional array computations. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 33–39, 2017. doi: 10.1145/3091966.3091973. URL http://doi.acm.org/10.1145/3091966.3091973.

G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. H. H. Juurlink. An evaluation of current SIMD programming models for C++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing, WPMVP@PPoPP 2016, Barcelona, Spain, March 13, 2016*, pages 3:1–3:8, 2016. doi: 10.1145/2870650.2870653. URL https://doi.org/10.1145/2870650.2870653.

S. Pop. *The SSA representation framework: semantics, analyses and GCC implementation.* PhD thesis, École Nationale Supérieure des Mines de Paris, 2006.

S. Pop, A. Cohen, and G. Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers, First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005, Proceedings*, pages 218–232, 2005. doi: 10.1007/11587514_15. URL https://doi.org/10.1007/11587514_15.

V. Porpodas and T. M. Jones. Throttling automatic vectorization: When less is more. In *2015 International Conference on Parallel Architectures and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 432–444, 2015. doi: 10.1109/PACT.2015.32. URL https:

//doi.org/10.1109/PACT.2015.32.

V. Porpodas, A. Magni, and T. M. Jones. PSLP: padded SLP automatic vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 190–201, 2015. doi: 10.1109/CGO.2015.7054199. URL https://doi.org/10.1109/CGO.2015.7054199.

R. T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138. ACM, 1959.

J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530, 2013. doi: 10.1145/2491956.2462176. URL https://doi.org/10.1145/2491956.2462176.

V. G. Reddy. Neon technology introduction. http://caxapa.ru/thumbs/301908/AT_-_NEON_for_Multimedia_Applications.pdf, 2008. Accessed: 2019.08.09.

O. Reiche. *A Domain-Specific Language Approach for Designing and Programming Heterogeneous Image Systems*. PhD thesis, University of Erlangen-Nuremberg, Germany, 2018. URL http://www.dr.hut-verlag.de/978-3-8439-3726-9.html.

N. Reissmann, T. L. Falch, B. A. Bjørnseth, H. Bahmann, J. C. Meyer, and M. Jahre. Efficient control flow restructuring for gpus. In *International Conference on High Performance Computing &amp; Simulation, HPCS 2016, Innsbruck, Austria, July 18-22, 2016*, pages 48–57, 2016. doi: 10.1109/HPCSim.2016.7568315. URL https://doi.org/10.1109/HPCSim.2016.7568315.

B. Ren, Y. Jo, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. Efficient execution of recursive programs on commodity vector hardware. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 509–520, 2015. doi: 10.1145/2737924.2738004. URL http://doi.acm.org/10.1145/2737924.2738004.

B. Rodrigues, F. M. Q. Pereira, and D. F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 110–120, 2016. doi: 10.1145/2892208.2892230. URL https://doi.org/10.1145/2892208.2892230.

P. K. Romano, N. E. Horelik, B. R. Herman, A. G. Nelson, B. Forget, and K. Smith. Openmc: A state-of-the-art monte carlo code for research and development. *Annals of Nuclear Energy*, 82:90 – 97, 2015. ISSN 0306-4549. doi: https://doi.org/10.1016/j.anucene.2014.07.048. URL http://www.sciencedirect.com/science/article/pii/S030645491400379X. 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.

J. Rosemann, S. Moll, and S. Hack. An abstract interpretation for SPMD divergence on reducible control flow graphs. *Proc. ACM Program. Lang.*, 5(POPL):1–31, 2021. doi: 10.1145/3434312. URL https://doi.org/10.1145/3434312.

N. Rotem. Intel OpenCL implicit vectorization module. In *LLVM Developer Meeting*, 2011.

N. Rotem and Y. Ben-Asher. Block unification if-conversion for high performance architectures. *Computer Architecture Letters*, 13(1):17–20, 2014. doi: 10.1109/L-CA.2012.28. URL https://doi.org/10.1109/L-CA.2012.28.

D. Sampaio, R. M. de Souza, S. Collange, and F. M. Q. Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13:1–13:36, 2013. doi: 10.1145/2523815. URL http://doi.acm.org/10.1145/2523815.

D. N. Sampaio, L. Pouchet, and F. Rastello. Simplification and runtime resolution of data dependence constraints for loop transformations. In *Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017*, pages 10:1–10:11, 2017. doi: 10.1145/3079079.3079098. URL https://doi.org/10.1145/3079079.3079098.

V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In *Languages and Compilers for Parallel Computing, 5th International Workshop, New Haven, Connecticut, USA, August 3-5, 1992, Proceedings*, pages 16–30, 1992. doi: 10.1007/3-540-57502-2_37. URL https://doi.org/10.1007/3-540-57502-2_37.

T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 249–265, 2017. URL http://dl.acm.org/citation.cfm?id=3018758.

S. Schneider, G. Smolka, and S. Hack. A linear first-order functional intermediate language for verified compilers. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, pages 344–358, 2015. doi: 10.1007/978-3-319-22102-1_23. URL https://doi.org/10.1007/978-3-319-22102-1_23.

M. Sharir. Structural analysis: A new approch to flow analysis in optimizing compilers. *Comput. Lang.*, 5(3):141–153, 1980. doi: 10.1016/0096-0551(80)90007-7. URL https://doi.org/10.1016/0096-0551(80)90007-7.

N. Shibata et al. SLEEF vector math libary. https://sleef.org/, 2019. Accessed: 2019.08.19.

J. Shin. Introducing control flow into vectorized code. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), Brasov, Romania, September 15-19, 2007*, pages 280–291, 2007. doi: 10.1109/PACT.2007.41. URL http://doi.ieeecomputersociety.org/10.1109/PACT.2007.41.

J. Shin, M. W. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*, pages 165–175, 2005. doi: 10.1109/CGO.2005.33. URL https://doi.org/10.1109/CGO.2005.33.

J. Shin, M. W. Hall, and J. Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. *Microprocessors and Microsystems - Embedded Hardware Design*, 33(4):235–243, 2009. doi: 10.1016/j.micpro.2009.02.002. URL https://doi.org/10.1016/j.micpro.2009.02.002.

B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '90, Island of Crete, Greece, July 2-6, 1990*, pages 350–359, 1990. doi: 10.1145/97444.97702. URL https://doi.org/10.1145/97444.97702.

D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 327–339, 2018. doi: 10.1145/3168812. URL https://doi.org/10.1145/3168812.

N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Prémillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2):26–39, 2017. doi: 10.1109/MM.2017.35. URL https://doi.org/10.1109/MM.2017.35.

M. Steuwer, T. Remmelg, and C. Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, pages 74–85, 2017. URL http://dl.acm.org/citation.cfm?id=3049841.

A. Stoutchinin and F. de Ferrière. Efficient static single assignment form for predication. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 172–181, 2001. doi: 10.1109/MICRO.2001.991116. URL https://doi.org/10.1109/MICRO.2001.991116.

J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. W. Hwu. Efficient compilation of fine-grained spmd-threaded programs for multicore cpus. In *Proceedings of the CGO 2010, The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, April 24-28, 2010*, pages 111–119, 2010. doi: 10.1145/1772954.1772971. URL http://doi.acm.org/10.1145/1772954.1772971.

J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

H. Sun, F. Fey, J. Zhao, and S. Gorlatch. WCCV: improving the vectorization of if-statements with warp-coherent conditions. In *Proceedings of the ACM International Conference on Supercomputing, ICS 2019, Phoenix, AZ, USA, June 26-28, 2019*, pages 319–329, 2019. doi: 10.1145/3330345.3331059. URL https://doi.org/10.1145/3330345.3331059.

T. Tholey. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theor. Comput. Sci.*, 465:35–48, 2012. doi: 10.1016/j.tcs.2012.09.025. URL https://doi.org/10.1016/j.tcs.2012.09.025.

S. Timnat, O. Shacham, and A. Zaks. Predicate vectors if you must. In *Workshop on Programming Models for SIMD/Vector Processing*, 2014.

J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. 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.

K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 327–337, 2009. doi: 10.1109/PACT.2009.18. URL `https://doi.org/10.1109/PACT.2009.18`.

P. Tu and D. A. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*, pages 414–423, 1995a. doi: 10.1145/224538.224648. URL `http://doi.acm.org/10.1145/224538.224648`.

P. Tu and D. A. Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, pages 47–55, 1995b. doi: 10.1145/207110.207115. URL `https://doi.org/10.1145/207110.207115`.

F. Wahlster. Vectorising divergent control-flow for simd applications. Master's thesis, Technische Universität München, 2018. URL `https://github.com/rAzoR8/EuroLLVM19/raw/master/docs/VectorizingDivergentControl-FlowforSIMDApplications-FabianWahlster.pdf`. Accessed: 2019.09.10.

I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, 2014. doi: 10.1145/2601097.2601199. URL `http://doi.acm.org/10.1145/2601097.2601199`.

N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 290–299, 1993. doi: 10.1145/155090.155118. URL `https://doi.org/10.1145/155090.155118`.

D. Wasserrab, D. Lohner, and G. Snelting. On pdg-based noninterference and its modular proof. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 31–44, 2009. doi: 10.1145/1554339.1554345. URL `https://doi.org/10.1145/1554339.1554345`.

C. Wimmer and H. Mössenböck. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, pages 132–141, 2005. doi: 10.1145/1064979.1064998. URL `http://doi.acm.org/10.1145/1064979.1064998`.

H. Wu, G. F. Diamos, J. Wang, S. Li, and S. Yalamanchili. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *IJHPCA*, 26(2):170–185, 2012. doi:

10.1177/1094342011434814. URL https://doi.org/10.1177/1094342011434814.

P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1993.

C. Yount. Vector folding: Improving stencil performance via multi-dimensional simd-vector representation. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*, pages 865–870, 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.27. URL https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27.

F. Yue, J. Pang, J. Jin, and D. Chao. Convergence and scalarization in whole function vectorization. In *IEEE 11th International Conference on Dependable, Autonomic and Secure Computing, DASC 2013, Chengdu, China, December 21-22, 2013*, pages 536–539, 2013. doi: 10.1109/DASC.2013.120. URL https://doi.org/10.1109/DASC.2013.120.

J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 427–440, 2012. doi: 10.1145/2103656.2103709. URL https://doi.org/10.1145/2103656.2103709.

H. Zhou and J. Xue. Exploiting mixed SIMD parallelism by reducing data reorganization overhead. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, pages 59–69, 2016. doi: 10.1145/2854038.2854054. URL https://doi.org/10.1145/2854038.2854054.