



An Abstract Interpretation for SPMD Divergence on Reducible Control Flow Graphs

JULIAN ROSEMANN, Saarland University, Saarland Informatics Campus, Germany
SIMON MOLL*, NEC Deutschland, Germany
SEBASTIAN HACK, Saarland University, Saarland Informatics Campus, Germany

Vectorizing compilers employ divergence analysis to detect at which program point a specific variable is *uniform*, i.e. has the same value on all SPMD threads that execute this program point. They exploit uniformity to retain branching to counter branch divergence and defer computations to scalar processor units. Divergence is a hyper-property and is closely related to non-interference and binding time. There exist several divergence, binding time, and non-interference analyses already but they either sacrifice precision or make significant restrictions to the syntactical structure of the program in order to achieve soundness.

In this paper, we present the first abstract interpretation for uniformity that is general enough to be applicable to reducible CFGs and, at the same time, more precise than other analyses that achieve at least the same generality.

Our analysis comes with a correctness proof that is to a large part mechanized in Coq. Our experimental evaluation shows that the compile time and the precision of our analysis is on par with LLVM's default divergence analysis that is only sound on more restricted CFGs. At the same time, our analysis is faster and achieves better precision than a state-of-the-art non-interference analysis that is sound and at least as general as our analysis.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Software verification and validation*; • **Computer systems organization** → *Single instruction, multiple data*.

Additional Key Words and Phrases: Divergence Analysis, Vectorization, Binding Time, Hyper-Property, Dependence, Non-Interference, Abstract Interpretation

ACM Reference Format:

Julian Rosemann, Simon Moll, and Sebastian Hack. 2021. An Abstract Interpretation for SPMD Divergence on Reducible Control Flow Graphs. *Proc. ACM Program. Lang.* 5, POPL, Article 31 (January 2021), 31 pages. <https://doi.org/10.1145/3434312>

1 INTRODUCTION

Vectorization is crucial to achieve performance on SPMD¹ (i.e. data-parallel) programs. SPMD code emerges from dedicated single program multiple data (SPMD) languages such as CUDA or ISPC as well as in compiler-driven loop vectorization. A SPMD program executes a given scalar program in multiple instances (often called lanes or threads). Vectorizing compilers employ divergence analysis to detect at which program point a specific variable is *uniform*, i.e. has the same value on all

*work partly done while at Saarland University

¹single program, multiple data

Authors' addresses: Julian Rosemann, Saarland University, Saarland Informatics Campus, Computer Science, Campus E1 3, 66123, Saarbrücken, Saarland, Germany; Simon Moll, NEC Deutschland, Stuttgart, Germany; Sebastian Hack, Saarland University, Saarland Informatics Campus, Computer Science, Campus E1 3, 66123, Saarbrücken, Saarland, Germany.

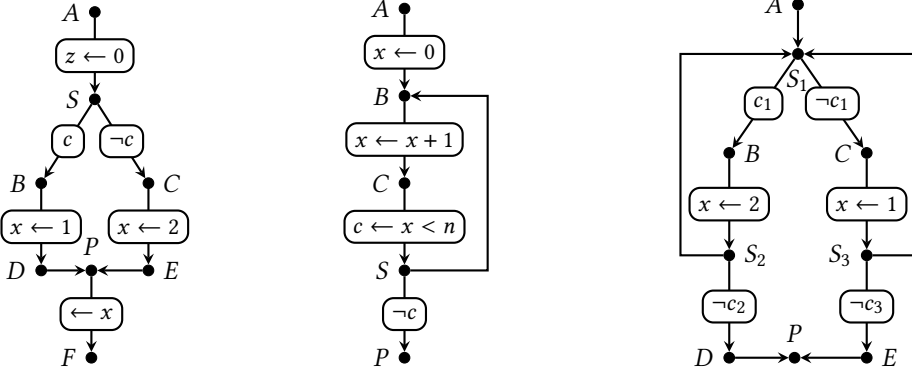


This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/author(s).

2475-1421/2021/1-ART31

<https://doi.org/10.1145/3434312>



(a) x is uniform at D and E but it depends on c 's uniformity, if x is uniform at P .

(b) x is uniform everywhere in the loop. x is uniform at P only if n is uniform.

(c) A more complicated example with three loop splits. All three branch variables c_1, c_2, c_3 need to be uniform for x to be uniform.

Fig. 1. Different situations in which a variable is uniform.

instances that execute this program point. They exploit uniformity in order to defer computations to scalar processor units and fight branch divergence by retaining branching.

Note that the term *divergence* comes from two different lanes taking different targets upon a branch whose predicate is *varying*, i.e. non-uniform. Therefore the pairs of terms *uniform*, *varying* and *non-divergent*, *divergent* are sometimes used interchangeably in the literature. In this paper, we use the term *uniformity* to emphasize that the goal of the analysis is to identify uniform variables. Branch divergence is merely a consequence of non-uniform branch variables.

To develop a better intuition of uniformity, consider Figure 1a. Variable x is uniform at program point D (and E) because the states at program point D (respectively E) of all traces that reach it agree on the value of variable x . At program point P this is no longer the case under the assumption that some traces reach P via D and some via E . However, if the branch condition c is uniform (at S) as well, all traces reach P either via D or E and x will be uniform at P .

Similarly, in Figure 1b, variable x is uniform at all program points inside the loop because, for every iteration of the loop, the states that appear in the particular traces agree on x for each program point. For example, the states of all traces that reach S in the same loop iteration agree on the value of x . At program point P however, x is not uniform if there are two traces that exit the loop after a different number of iterations. This is exactly the case if the loop exit condition c is not uniform.

Note that being uniform does not imply being constant. Reconsider Figure 1a. If c is uniform, we do not know if all traces that reach A will go left to B or right to C . What we know is that either all of them go left or all of them go right.


From an abstract interpretation [Cousot and Cousot 1977] point of view, divergence analysis is not a “standard” value analysis (such as constant propagation, intervals, global value numbering, etc.). A value analysis provides invariants on states that hold for each execution *individually*. For example, if an interval analysis provides the invariant $z \in [1, 5]$ for some program point p , this means that for every single execution of the program, variable z has a value between 1 and 5 every time the execution visits p . Uniformity however does not talk about single executions in isolation but relates different executions: x being uniform means that two *different* executions of the same program agree on x 's value at a specific point in their execution. In that sense, uniformity is a

hyper-property [Clarkson and Schneider 2010] and is similar to hyper-properties such as non-interference in information flow analysis and binding time in partial evaluation.²

As discussed by means of Figure 1, the crucial point in divergence analysis is to get the abstract transformers for control flow joins and loop exits right by identifying the right set of control-flow split nodes that influence their uniformity. Existing analyses simplify this problem in various ways: by imposing significant restrictions on the syntactical structure of the program such as supporting only structured syntax (if-then-else, while loops), or by forbidding branches with more than two successors, for example. Most formal treatments of divergence analysis rely on inductively-defined structured syntax. In this setting, the transformers can be formulated *locally* because all necessary ingredients (branch/loop condition and the statements in the body of the control structure) are located in the same syntactical element. Other approaches resort to program representations, such as the gated SSA (GSA) form, that augment the program by ascribing the branch predicates of all the relevant split nodes to the respective join nodes. However, these representations are rarely used in practice and do not come with formalized or even verified construction algorithm which puts the soundness of divergence analyses that build on them at risk. Finally, several approaches that are also used in production code use ad-hoc techniques (often involving some sort of control-dependence criterion) that are unnecessarily imprecise or only sound under significant syntactical restrictions that are not always clearly documented.

In summary, existing approaches are either not treated formally, imprecise, or make restrictions on control flow that impedes their use in practice (e.g. LLVM-based GPU drivers) considerably (we discuss related work in more detail in Section 8).

Contributions. In this paper, we present, to the best of our knowledge, the first formal account of an abstract interpretation of uniformity on reducible control flow graphs (CFGs). Requiring reducibility³ is the only restriction we make on the CFG. In summary, we make the following contributions:

- We formalize the intuition about divergence that we uniformity in this section based on a novel trace semantics that augments a standard trace semantics by input-independent *tags*. These tags uniquely identify instances of program points and permit to *relate* program point instances across different executions in a meaningful way (Section 3.2).
- Based on this trace semantics, we formalize abstract domains and transformers (Section 4) and prove their soundness. The correctness of our analysis is supported by an accompanying Coq development⁴ that mechanizes a large part of the proofs. We indicate the parts currently formalized in Coq with the Coq logo .
- We derive a new criterion of branch divergence directly from the correctness proof of the abstract transformer (Section 4.3). This criterion is based on identifying nodes of disjoint paths on a specific directed acyclic graph (DAG) that is derived from the program's CFG. For the problem of identifying joins of disjoint paths in a DAG we present a novel, simple, and optimal $O(|V| + |E|)$ algorithm (Section 5).
- We also give a completeness theorem (Section 4.6) that shows that our analysis identifies no more than the relevant split nodes (up to the precision of the abstract edge effects and semantically infeasible paths).

²In non-interference, uniform and varying correspond to *public* and *private* (or *low* and *high*). In binding-time analysis they are called *static* and *dynamic*.

³By folk wisdom, an overwhelming majority of practically-relevant CFGs are reducible.

⁴The development can be found at <https://github.com/cdl-saarland/uniana>. See also Appendix B.1.

- We compare our analysis against two state-of-the-art analyses by experimentally evaluating the analysis time and the precision (in terms of instructions classified uniform) in the context of SPMD vectorization on a wide range of benchmarks sets. The first analysis is LLVM’s default divergence analysis [Coutinho et al. 2011] that is only sound under syntactical restrictions stronger than reducibility. The second, a control-dependence-based non-interference analysis [Wasserrab et al. 2009], is applicable to reducible CFGs, but is less precise than our analysis.

In terms of compile time, our analysis is on par with LLVM’s analysis and achieves a speedup of 25% over the second analysis. In terms of precision, our analysis is equally precise as LLVM’s and classifies 5% more instructions and branches as uniform as the second. Finally, no benchmark we encountered contained an irreducible CFG, which supports the practicality of our algorithm.

Structure of this paper. The next section introduces the key concepts of this paper informally and provides intuition for our analysis. Section 3 reviews basic definitions on CFGs and defines the semantics we use. Section 4 is the core part of the paper that defines the abstract interpretation and proves the abstract transformer correct. Section 6 presents an add-on to the analysis that provides more precision when the program is not under static single assignment (SSA) form. Section 5 introduces a novel algorithm for identifying disjoint paths and proves it correct. Section 7 evaluates the analysis on several OpenCL benchmarks and tree-traversal kernels and compares the analysis runtime with the state-of-the-art divergence analysis implemented in LLVM. Finally, Section 8 discusses related work.

2 OVERVIEW

In this section, we review basic concepts and discuss the core elements of our analysis informally.

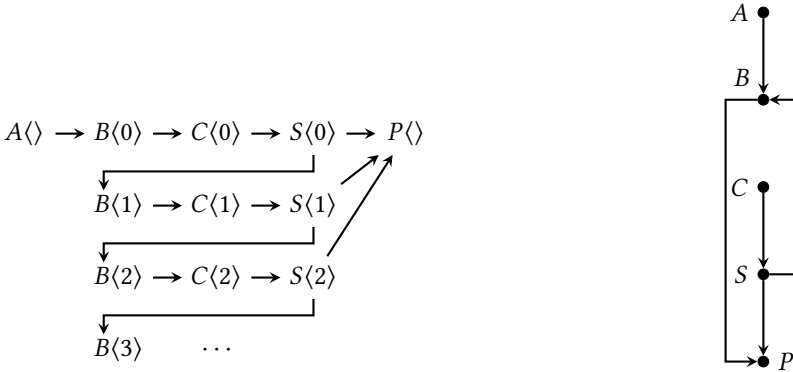
2.1 Relating States in Different Traces using Tags

We model a single execution of some SPMD program P by a set of traces T where each trace $t \in T$ corresponds to the execution of an *instance* of this program. (We will use the term *thread* for an instance of a program to not confound it with *instances* of program points.) The set of all SPMD executions of P is therefore given by a set of sets of traces \mathfrak{T} . Given a set of sets of states \mathfrak{S} , a variable x is called uniform on \mathfrak{S} if for each set of program states $S \in \mathfrak{S}$ and each pair of states $\{\sigma, \sigma'\} \subseteq S$, there is $\sigma x = \sigma' x$. To lift this notion of state-based uniformity to program executions, we need to be able to identify individual pairs of states in pairs of traces from one SPMD execution of the program.

One way of doing this is to relate states that appear at the same program point. However, since a program point can appear multiple times in a trace, program points alone are inadequate to uniquely identify pairs of states in traces. To this end, we equip the semantics with an input-independent part that *tags* each configuration (element of a trace) with the values of the induction variables of all loops that surround the program point of the configuration. This way, each program point instance can be uniquely identified. We write the pair of program point P and a tag i as $P\langle i \rangle$ and call it an *instance* of P . This is also the reason why we require the CFGs to be reducible: only then is there a well-defined notion of loop nesting on which our definition of tagging relies on.

This tagging gives rise to the *tagged* CFG, a potentially⁵ infinite DAG whose nodes consist of the program point instances. Figure 2a shows the tagged CFG for the example program in Figure 1b.

⁵The tagged CFG is finite if and only if the CFG does not contain loops.



(a) The tagged CFG for Figure 1b is an infinite DAG that consists of all possible program point instances. Local inhomogeneity at $P\langle \rangle$ is witnessed by an instance with multiple successors (here e.g. $S\langle 0 \rangle$) and two disjoint paths from that instances to $P\langle \rangle$.

(b) The head-rewired CFG for Figure 1b emerges from the CFG by deleting all outgoing edges of each loop header (here $B \rightarrow C$) and adding edges from each header to its exit (here $B \rightarrow P$). For each pair of instance-disjoint sub-traces that split at $s\langle k \rangle$ and join at $p\langle i \rangle$, there are two node-disjoint sub-paths splitting at s and rejoining at p in the head-rewired CFG. Here, e.g.: $S\langle 0 \rangle, P\langle \rangle$ and $S\langle 0 \rangle, B\langle 1 \rangle, C\langle 1 \rangle, S\langle 1 \rangle, P\langle \rangle$ in the tagged CFG correspond to S, P and S, B, P .

Fig. 2. The tagged CFG and the head-rewired CFG for Figure 1b.

2.2 An Abstract Interpretation for Uniformity

Tags allow us to appropriately define what “variable x is uniform at a program point P ” means: For each pair of traces $t, t' \in T$ and each tag i : If t contains a configuration $(P\langle i \rangle, \sigma)$ and t' contains a configuration $(P\langle i \rangle, \sigma')$, then $\sigma x = \sigma' x$. This is the informal version of the concretization of the uniformity abstract domain we present in Section 4.1.

Defining abstract transformers for the effects on control flow edges is straightforward for uniformity. For example, a sound abstract transformer for the edge effect $z \leftarrow x + y$, yields that x, y, z are uniform (after executing the statement) if x and y are uniform (before).

The interesting part of the analysis is the transformer that computes the abstract information for program points. Reconsider Figure 1a. Intuitively, the uniformity of x depends on the uniformity of c . To see this, assume that c is not uniform at S . Then, there are two traces⁶

$$\begin{aligned} t &= \dots (S\langle \rangle, \sigma), (B\langle \rangle, \cdot), (D\langle \rangle, \cdot), (P\langle \rangle, \sigma_P), \dots \\ t' &= \dots (S\langle \rangle, \sigma'), (C\langle \rangle, \cdot), (E\langle \rangle, \cdot), (P\langle \rangle, \sigma'_P), \dots \end{aligned} \tag{1}$$

for which $\sigma c \neq \sigma' c$. Note that $\langle \rangle$ denotes the tag for nodes that are not in a loop. One of the traces “goes left” while the other one “goes right”. It is important to understand that x is nevertheless uniform at D because all traces that reach D (under the same tag) agree on the value of x . (The same argument applies to E .) However, both traces disagree on x at P because $\sigma_P x = 1$ and $\sigma'_P x = 2$. So, x is not uniform at P although it is uniform at D and E .

Hence, in contrast to “common” value analyses, we cannot just compute the uniformity of a variable at a program point by “joining” the uniformity information of that variable at its predecessors.

⁶The dot \cdot means that the state is irrelevant for the example.

The gist is that we can only leverage uniformity at the predecessors of P if the traces behave *locally homogeneously* which means that each instance $P\langle i \rangle$ is preceded by the same instance $q\langle j \rangle$ of some predecessor q in every trace.⁷ This, combined with the fact that x is uniform at $D\langle \cdot \rangle$ and $E\langle \cdot \rangle$ implies that x has the same value on all traces that reach $P\langle \cdot \rangle$. Note that because there is no loop in this example, the tags of all instances are the same.

This changes in Figure 1b. P has only one control flow predecessor S . If however n is not uniform at S , there may be two traces that exit the loop in different iterations, leading to two differently-tagged instances of S preceding $P\langle \cdot \rangle$ (This can be nicely seen in the tagged CFG in Figure 2a):

$$\begin{aligned} t &= \dots, (A\langle \cdot \rangle, \sigma), (B\langle 0 \rangle, \cdot), (C\langle 0 \rangle, \cdot), (S\langle 0 \rangle, \cdot), (P\langle \cdot \rangle, \sigma_P), \dots \\ t' &= \dots, (A\langle \cdot \rangle, \sigma), (B\langle 0 \rangle, \cdot), (C\langle 0 \rangle, \cdot), (S\langle 0 \rangle, \cdot), \\ &\quad (B\langle 1 \rangle, \cdot), (C\langle 1 \rangle, \cdot), (S\langle 1 \rangle, \cdot), (P\langle \cdot \rangle, \sigma'_P) \dots \end{aligned} \quad (2)$$

Note that x is uniform at S because for every tag i , all traces that contain $S\langle i \rangle$ agree on x at $S\langle i \rangle$. Nevertheless we have $\sigma_P x = 1$ and $\sigma'_P x = 2$ and x is not uniform at P : Although S directly precedes P in every trace that contains $P\langle \cdot \rangle$, it does so with different instances $S\langle 0 \rangle$ and $S\langle 1 \rangle$.

2.3 Split Points and Disjoint Traces

If an instance of a program point is preceded by two different instances (i.e. it is locally inhomogeneous), these two different instances can be extended to two *instance-disjoint* sub-traces. Both of these sub-traces originate in an instance that appears in both traces (note that all traces start with the same instance $rt\langle \cdot \rangle$). So, local inhomogeneity at some instance $p\langle i \rangle$ is witnessed by a *split instance* $s\langle k \rangle$ and two disjoint sub-paths in the *tagged CFG*.

Reconsider the traces in (2). Both traces correspond to two paths in the tagged CFG in Figure 2a. The corresponding split instance is $S\langle 0 \rangle$. If the traces were such that the first trace left the loop after 4 iterations and the second after 17 iterations, the respective split instance would be $S\langle 4 \rangle$.

This split instance is the instance of a *split node*, i.e. a node with multiple control-flow successors. We will show later that at this instance of the split point both traces *diverge*, i.e. go to different successor program points. This can only happen if the branch predicate of the split is *varying*, i.e. takes different boolean values on each trace. Conversely, if the branch predicate of every such split point is *uniform*, there will be no diverging traces and this split will not cause local inhomogeneity at other program points.

The core of the uniformity analysis presented in this paper is to identify the split points that are *relevant* for local homogeneity at some program point p . The main result is that there is a simple way to transform the CFG of the program into a *finite* DAG that we call *head-rewired CFG*, such that for each pair of *instance-disjoint traces* (i.e. disjoint sub-paths in the tagged CFG) that originate from a split s and a rejoin at some node p , there are two disjoint paths in the head-rewired CFG and vice versa (Section 4.6).

Therefore, identifying all joins for which s is relevant boils down to identifying all program points where two disjoint paths in the head-rewired CFG rejoin. (Note that the disjoint paths start at the successors of s in the CFG.) For computing these joins, we present a simple, efficient, and easy-to-implement algorithm (Section 5).

The head-rewired CFG results from the program's CFG by replacing the outgoing edges of every loop header by edges to the exits of the corresponding loop. Figure 2b shows the DAG for Figure 1b.

⁷We use capitals for concrete nodes and lower-case letters for variables that range over nodes.

The intuition behind the head-rewired CFG is the following: The instance-disjoint sub-traces in (2) are caused by a split inside a loop that causes one trace to leave the loop and the other to remain in the loop. Such two sub-traces do not necessarily correspond to two *node*-disjoint paths in the CFG (like in this example). The reason is that the trace that remains in the loop may visit the same nodes as the other trace, however with a “later” tag which makes the traces (instance-)disjoint but not the paths. Therefore, replacing the outgoing edges of the header with edges that lead to the loop exits creates node-disjoint paths for such instance-disjoint traces which in turn makes the disjoint sub-traces identifiable as disjoint paths.

Replacing these edges is compatible with splits that create instance-disjoint traces that *do* correspond to node-disjoint paths such as the one in (1). On their way to their common join point, such disjoint paths can enter loops but also have to exit them which is possible via the replaced edge (because it leads to an exit). It is not possible that both paths need to enter loops (without exiting them) to reach their join point because loop headers dominate all nodes in a loop and would forcibly be contained on both paths which contradicts their disjointness.

In Section 4.6 we show that the set of split nodes in the head-rewired CFG is equivalent to the set of split nodes in the *tagged* CFG.

3 PROGRAMS

3.1 Control Flow Graphs

We consider a program to be given by a control flow graph (CFG) $G = (\mathbf{Lab}, \rightarrow, rt)$ that consists of nodes (also called labels or program points) \mathbf{Lab} , edges $\rightarrow \subseteq \mathbf{Lab} \times \mathbf{Lab}$ and a unique root node $rt \in \mathbf{Lab}$ that has no incident edges. A *path* is a sequence of nodes that are connected by edges. We will use the notation $p \rightarrow^* q$ for a path from p to q . A *split point* is a node with more than one control flow successor.

Reducible CFGs. In this paper, we restrict ourselves to *reducible* [Hecht and Ullman 1974] CFGs. In practice, this is not a severe constraint because almost every CFG encountered in practice is reducible. In a reducible CFG, each edge whose target dominates its source is called a *back edge*. Its target is the *header* of a loop and its source is called a *latch*. The *loop* of some header h is a set of nodes that consists of all nodes that are dominated by h and from which a latch of h is reachable along a path that does not contain h itself. For two different loops it holds that they are either disjoint or one is contained in the other. Therefore, the loops of a reducible CFG form a forest that can be turned into a tree by inserting an artificial root. For the sake of simplicity, we assume an artificial top-level loop that spans the entire CFG.

We identify loops with their headers, thus for a loop header h and node v the statement $v \in h$ means that v is a member of h 's loop. The loop of a node v is defined to be the innermost loop that contains v and is written as h_v . The *depth* of a loop is the distance of the loop to the root in the loop tree. For nodes p and q , if $p \notin h_q$ and $q \rightarrow p$, we say that q is a *loop exiting node*, p is a *loop exit* of h_q , and $q \rightarrow p$ is an *exit edge*. Similarly, if $q \notin h_p$ then $q \rightarrow p$ is an *entry edge*.

Normalizations. We normalize the CFG to simplify the formal treatment by imposing the following requirements:

- (1) All predecessors of an exit node are in the same loop.
- (2) The target of a loop exit edge is not a loop header.
- (3) There is no edge from a node to itself.
- (4) An exit edge can only exit one loop at a time.

All of these requirements can be easily implemented: Requirements 1, 2, and 3 are met by splitting the corresponding edges with an additional node per edge. We obey Requirement 4 as follows: We

split each exit edge that spans several loops by introducing a node per bypassed loop. We place each such node into its loop by adding a “fake” back edge to the header of its loop. Consequently, this makes such a node a split node. We assign the exiting edge the effect **true** and the back edge the effect **false** which makes the branch variable of this split *uniform*.

3.2 Tags

As indicated in Section 2.1, we equip every configuration (element) of a program trace with an input-independent *tag* that identifies the configuration uniquely within that trace. We define the tag of a configuration to be an *iteration vector*, i.e. the valuation of the induction variables of all loops that surround the program point of the configuration. Hence, the set of tags is defined as the set of sequences of natural numbers:

$$\mathbf{Tag} \triangleq \mathbb{N}^*$$

We define $|i|$ as the *length* of $i \in \mathbf{Tag}$. The type of an edge $e = q \rightarrow p$ determines its tag semantics g_{qp} .

✦ **DEFINITION 3.1** (TAG SEMANTICS).

$$\begin{array}{ll} g_{qp} \langle i_1 \dots i_n \rangle \triangleq \langle i_1 \dots i_n 0 \rangle & \text{if } e \text{ is an entry edge} \\ g_{qp} \langle i_1 \dots i_n i_{n+1} \rangle \triangleq \langle i_1 \dots i_n (i_{n+1} + 1) \rangle & \text{if } e \text{ is a back edge} \\ g_{qp} \langle i_1 \dots i_n i_{n+1} \rangle \triangleq \langle i_1 \dots i_n \rangle & \text{if } e \text{ is an exit edge} \\ g_{qp} \langle i_1 \dots i_n \rangle \triangleq \langle i_1 \dots i_n \rangle & \text{otherwise} \end{array}$$

Note that this case distinction is exclusive because we require the CFG to be reducible. We define the initial tag to be the empty tuple $\langle \rangle$.

✦ **DEFINITION 3.2** (TAG ORDERING). For each $n \in \mathbb{N}$, we define the (total) ordering \leq_n of all tags in \mathbb{N}^n to be the lexicographic order with innermost loop dimensions being least significant. We further define $\leq \triangleq \bigcup_{n \in \mathbb{N}} \leq_n$.

Based on the tag semantics, we define the (possibly) infinite graph of program point instances:

✦ **DEFINITION 3.3** (TAGGED CFG). The tagged CFG is the rooted graph $G_T \triangleq (V_T, \rightarrow_T, rt\langle \rangle)$:

$$\begin{array}{l} V_T \triangleq \{p\langle i \rangle \in \mathbf{Lab} \times \mathbf{Tag} \mid \text{depth } p = |i|\} \\ \rightarrow_T \triangleq \{(q\langle j \rangle, p\langle i \rangle) \in V_T^2 \mid q \rightarrow p \wedge g_{qp} j = i\}. \end{array}$$

Analogously to \rightarrow^* we use \rightarrow_T^* for paths on G_T .

Since every back edge traversal increases the tag, G_T is a DAG:

✦ **LEMMA 3.4** (ACYCLICITY). G_T is acyclic.

We define *preceding instances* to define a (stricter) notion of dominance on G_T .

✦ **DEFINITION 3.5** (PRECEDING INSTANCES). We say an instance $q\langle j \rangle$ precedes an instance $p\langle i \rangle$ on a G_T path

$$rt\langle \rangle \dots, q\langle j \rangle, \underbrace{\dots, p\langle i \rangle, \dots}_r$$

if there is no j' such that $q\langle j' \rangle \in r$. We write $q\langle j \rangle \leq p\langle i \rangle$.

3.3 Semantics

For each program point q there is an effect function $c_q : \text{St} \rightarrow \text{Lab} \times \text{St}$ that decides, based on some *state* $\sigma \in \text{St} \triangleq \text{Var} \rightarrow \text{Val}$, where the computation shall continue and in what state. The continuation program point p must be a control-flow successor of q . We require that every split node s has an individual branch variable b_s such that the value of its effect function c_s only depends on b_s . The fact that c_q is a (partial) function ensures that branching is deterministic.⁸

An effect function c_q gives rise to *edge effect*⁹ functions f_{qp} that describe the transformation of states along each control flow edge $q \rightarrow p \in E$ in the following way:

$$f_{qp} \sigma = \sigma' \iff c_q \sigma = (p, \sigma') \quad \text{for all } \{\sigma, \sigma'\} \subseteq \text{St}$$

Computations are given by traces. A trace is a sequence of *configurations*, which are pairs of program point instances and states.

For a given set of initial states $S \subseteq \Sigma$, the semantics of a program is the least fix-point of the (monotone) function F_S (aka concrete transformer):

‡ **DEFINITION 3.6** (CONCRETE TRANSFORMER).

$$\begin{aligned} F_S : \mathcal{P}(\text{Tr}) &\rightarrow \mathcal{P}(\text{Tr}) \\ T &\mapsto \left\{ (rt\langle \rangle, \sigma_0) \mid \sigma_0 \in S \right\} \\ &\cup \underbrace{\left\{ \dots, (q\langle j \rangle, \sigma), (p\langle g_{qp} j \rangle, f_{qp} \sigma) \mid t \in T \wedge q \rightarrow p \wedge f_{qp} \sigma \text{ defined} \right\}}_t \end{aligned}$$

If we do not care about a particular set of start states, we omit the index S . Note that when omitting the states, every trace induces a path in the tagged CFG.

‡ **DEFINITION 3.7** (TAGGED PATHS FOR A SET OF TRACES).

$$\begin{aligned} tpaths : \mathcal{P}(\text{Tr}) &\rightarrow \mathcal{P}(V_T^*) \\ T &\mapsto \{ p_1\langle i_1 \rangle \rightarrow_T \dots \rightarrow_T p_n\langle i_n \rangle \mid (p_1\langle i_1 \rangle, \sigma_1), \dots, (p_n\langle i_n \rangle, \sigma_n) \in T \} \end{aligned}$$

3.4 Hyper-Semantics

The set $\mathcal{P}(\text{Tr})$ is called the set of *hyper-traces*. We lift F to hyper-traces and define the hyper-semantics as the least fixpoint of the function

‡ **DEFINITION 3.8** (HYPER-SEMANTICS).

$$\begin{aligned} \mathfrak{F}_{\mathfrak{E}} : \mathcal{P}(\mathcal{P}(\text{Tr})) &\rightarrow \mathcal{P}(\mathcal{P}(\text{Tr})) \\ \mathfrak{T} &\mapsto \{ \{ (rt\langle \rangle, \sigma) \mid \sigma \in S \} \mid S \in \mathfrak{E} \} \\ &\cup \{ F_{\emptyset}(T) \mid T \in \mathfrak{T} \} \end{aligned}$$

4 UNIFORMITY ANALYSIS

In this section, we define the uniformity abstract domain, its concretization (Section 4.1) and transformer and prove the latter sound with respect to splits on tagged paths (Section 4.3). Because the tagged CFG is potentially infinite, this does not directly yield an efficiently implementable transformer.

Therefore, the crucial step remaining is to show that every pair of disjoint paths on the tagged CFG relates to a pair of disjoint paths on the head-rewired CFG. We outline this proof in Section 4.4 and carry it out in detail in Section 4.5 and Section 4.6. Section 4.6 also includes a proof of the backwards direction (i.e. for every pair of disjoint paths in the head-rewired CFG there is a pair of

⁸Identifying uniform branches in programs with non-deterministic branching is futile.

⁹Edge effects are more common in abstract interpretation and more amenable to the formal development.

disjoint paths in the tagged CFG), which is not required for the soundness of the transformer but shows that we are not losing precision with the use of the head-wired CFG.

4.1 Abstract Domain and Concretization

First, let us consider the following abstract domain that provides information about the uniformity of a certain variable in sets of states.

$$\mathbf{UniS} \triangleq \mathbf{Var} \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

A variable x is supposed to be uniform if for any two states σ_1, σ_2 in some set $S \subseteq \mathbf{St}$ of states, its value is equal:

$$\begin{aligned} \gamma_{\mathbf{UniS}} : \mathbf{UniS} &\rightarrow \mathcal{P}(\mathcal{P}(\mathbf{St})) \\ \mathit{uni} &\mapsto \{S \in \mathcal{P}(\mathbf{St}) \mid \forall x \in \mathbf{Var}. \forall \{\sigma_1, \sigma_2\} \subseteq S. \\ &\quad \mathit{uni} x \Rightarrow \sigma_1 x = \sigma_2 x\} \end{aligned}$$

Ultimately, we would like to assess the uniformity of variables in program executions in a flow-sensitive way. This is reflected by the following abstract domain:

$$\mathbf{Uni} \triangleq \mathbf{Lab} \rightarrow \mathbf{UniS}$$

As discussed in Section 2.1, we therefore have to relate the states that appear in two separate traces at the program point in question. However, since a program point can appear multiple times in a trace, a program point alone is not sufficient to relate the states in an unambiguous way. The concretization of the uniformity domain takes this into account by incorporating *tags* (cf. Section 3.2): For a variable x to be uniform at p then means that for two executions, the values of x are equal in all states at p that are equally tagged:

✦ **DEFINITION 4.1** (CONCRETIZATION OF \mathbf{Uni}).

$$\begin{aligned} \gamma_{\mathbf{Uni}} : \mathbf{Uni} &\rightarrow \mathcal{P}(\mathcal{P}(\mathbf{Tr})) \\ \mathit{uni} &\mapsto \{T \in \mathcal{P}(\mathbf{Tr}) \mid \forall x \in \mathbf{Var}. \forall p \in \mathbf{Lab}. \forall i \in \mathbf{Tag}. \\ &\quad \forall \{t_1, t_2\} \subseteq T. \forall \sigma_1, \sigma_2 \in \mathbf{St}. \mathit{uni} p x \Rightarrow \\ &\quad (p\langle i \rangle, \sigma_1) \in t_1 \Rightarrow (p\langle i \rangle, \sigma_2) \in t_2 \Rightarrow \\ &\quad \sigma_1 x = \sigma_2 x\} \end{aligned}$$

✦ **THEOREM 4.2.** $\gamma_{\mathbf{Uni}}$ is meet-preserving.¹⁰

4.2 Local Homogeneity

In many common static analyses the abstract value of a program point is defined as the join of the abstract values of its incident edges. The examples discussed in the introduction (cf. Figure 1) suggest that this is not correct for uniformity analysis. In Section 2.2, we informally introduced the concept of *local homogeneity* of a program point p which says that on all traces, each instance of that program point is preceded by the same instance of the same predecessor. Before we turn to uniformity itself, in this section, we describe homogeneity formally and derive necessary conditions for it that we then employ in the uniformity transformer.

✦ **DEFINITION 4.3** (LOCAL HOMOGENEITY). For a set of tagged paths P and a node $p \in \mathbf{Lab}$,

$$\begin{aligned} \mathit{hom} P p &\triangleq \forall q_1, q_2, p \in \mathbf{Lab}. \forall j_1, j_2, i \in \mathbf{Tag}. \\ &\quad \cdots \rightarrow_T q_1\langle j_1 \rangle \rightarrow_T p\langle i \rangle \rightarrow_T \cdots \in P \Rightarrow \\ &\quad \cdots \rightarrow_T q_2\langle j_2 \rangle \rightarrow_T p\langle i \rangle \rightarrow_T \cdots \in P \Rightarrow \\ &\quad q_1 = q_2 \wedge j_1 = j_2. \end{aligned}$$

¹⁰Meet-preserving mappings are monotone and induce a Galois connection.

Consider a program point p , a tag i , and two tagged paths π_1 and π_2 that both contain $p\langle i \rangle$:

$$\begin{aligned} \pi_1 &= \cdots \rightarrow_T s\langle k \rangle \rightarrow_T \underbrace{u_1\langle l_1 \rangle \rightarrow_T \cdots \rightarrow_T q_1\langle j_1 \rangle}_{r_1} \rightarrow_T p\langle i \rangle \rightarrow_T \cdots \\ \pi_2 &= \cdots \rightarrow_T s\langle k \rangle \rightarrow_T \underbrace{u_2\langle l_2 \rangle \rightarrow_T \cdots \rightarrow_T q_2\langle j_2 \rangle}_{r_2} \rightarrow_T p\langle i \rangle \rightarrow_T \cdots \end{aligned} \quad (3)$$

Assume that the program is not locally homogeneous at p , i.e. $q_1\langle j_1 \rangle \neq q_2\langle j_2 \rangle$. Since $rt\langle \rangle$ is in every tagged path, π_1 and π_2 have a at least one *common instance*. Let $s\langle k \rangle$ be a¹¹ *last common instance* of π_1 and π_2 before $p\langle i \rangle$ where r_1 and r_2 are disjoint and at least one of them is non-empty.¹² Such an $s\langle k \rangle$ is a *witness* of inhomogeneity and we say that s is *relevant* for p . Because r_1 and r_2 are instance-disjoint, $s\langle k \rangle$ is a split instance on the tagged CFG and also a split node in the CFG:

✦ **LEMMA 4.4.** s is a split node.

PROOF. Because of $q_1\langle j_1 \rangle \neq q_2\langle j_2 \rangle$, r_1 and r_2 cannot both be empty. Since r_1 and r_2 are disjoint by assumption and since $p\langle i \rangle \notin r_1 \cup r_2$ by Lemma 3.4, there is either $u_1 \neq u_2$ or $l_1 \neq l_2$. Suppose $u_1 = u_2$. Then, the determinism of tag semantics (Definition 3.1) implies $l_1 = l_2$. Contradiction. \square

Each split node s has a distinct branch variable b_s whose value determines to which program point it branches (see Section 3.3). Because of Lemma 4.4 this variable is *not uniform* at s in (3). Conversely, if the branch variable of every split node that is *relevant* for p is *uniform*, then all traces at p are locally homogeneous. Therefore, homogeneity at p is determined by the uniformity of the branch variables of its relevant splits.

Relevance, as defined above, is a semantic property that certainly is undecidable. As a first step towards a decidable criterion for local homogeneity (which will be the focus from Section 4.5 onwards), we consider all possible tagged paths instead of only those that are projections of traces and thereby get a sound over-approximation:

✦ **DEFINITION 4.5.**

$$\mathit{splits}_T p \triangleq \{s \in V \mid \exists \pi, \phi, k, i, \text{ such that } \pi \text{ and } \phi \text{ are disjoint paths from } s\langle k \rangle \text{ to } p\langle i \rangle \text{ in } G_T\}.$$

✦ **LEMMA 4.6.** For all program points s and p and all $\text{uni} \in \text{Uni}$:

$$\bigwedge_{s \in \mathit{splits}_T p} \text{uni } s \ b_s \implies \forall T \in (F \circ \gamma_{\text{Uni}} \text{ uni}). \text{ hom}(t\text{paths } T) p$$

The definition of splits_T does not directly induce an efficient implementation because the tagged CFG is infinite. From Section 4.5 on, we will establish the *equivalence* between splits_T and a similar property splits (cf. Definition 4.18) on the *finite* head-rewired CFG (briefly explained in Section 2.3, formally defined in Section 4.6), which can be efficiently computed.

4.3 An Abstract Transformer for Uniformity

In this section, we define an abstract transformer of the uniformity domain and prove its soundness. First of all, we assume that for each effect on a CFG edge $q \rightarrow p$ there is given a correct abstract effect

$$f_{qp}^\# : \text{UniS} \rightarrow \text{UniS} \quad \text{with } (\mathfrak{f}_{qp} \circ \gamma_{\text{UniS}}) u \subseteq (\gamma_{\text{UniS}} \circ f_{qp}^\#) u \quad \text{for all } u \in \text{UniS} \quad (4)$$

¹¹There may be more than one instance with this property.

¹²If r_1 or r_2 is empty, then $q_1 = s$ and $u_1 = p$ or $q_2 = s$ and $u_2 = p$, respectively.

that computes which variables remain uniform under \mathfrak{f}_{qp} . Here, \mathfrak{f}_{qp} is the function that lifts the concrete edge transformer f_{qp} to sets of sets of states. We do not want to go into details about abstract edge effects because their definition (and soundness proof) is straightforward for the effects one is typically interested in (such as assignments). Therefore, we assume that for each edge, we are given a *sound* abstract edge transformer.

The uniformity abstract transformer makes use of homogeneity information we developed in the last section:

✦ **DEFINITION 4.7** (UNIFORMITY ABSTRACT TRANSFORMER).

$$F_{\text{Uni}}^{\#} : \text{Uni} \rightarrow \text{Uni}$$

$$uni \mapsto \lambda p. \lambda x. \underbrace{\bigwedge_{s \in \text{splits}_T p} uni \ s \ b_s}_{H_p} \wedge \bigwedge_{q \rightarrow p} f_{qp}^{\#} (uni \ q) \ x$$

✦ **THEOREM 4.8.** $F_{\text{Uni}}^{\#}$ is sound.

PROOF. We need to show, for all $uni \in \text{Uni}$:

$$(\mathfrak{F} \circ \gamma_{\text{Uni}}) \ uni \subseteq (\gamma_{\text{Uni}} \circ F_{\text{Uni}}^{\#}) \ uni$$

Consider some $uni \in \text{Uni}$, a set of traces $T \in (\mathfrak{F} \circ \gamma_{\text{Uni}}) \ uni$, and two traces $\{t_1, t_2\} \subseteq T$. Based on Definition 4.1, we need to show that for each instance $p\langle i \rangle$ and each variable x , if $(F_{\text{Uni}}^{\#} \ uni) \ p \ x = \mathbf{true}$ and the configurations $(p\langle i \rangle, \sigma_1) \in t_1$ and $(p\langle i \rangle, \sigma_2) \in t_2$ do exist, both agree on x , i.e. $\sigma_1 \ x = \sigma_2 \ x$.

So, consider a variable x , a program point p , a tag i and two states σ_1, σ_2 such that $(p\langle i \rangle, \sigma_1) \in t_1$ and $(p\langle i \rangle, \sigma_2) \in t_2$. Then, t_1, t_2 look like this:

$$\begin{aligned} t_1 &= \dots \ (q_1\langle j_1 \rangle, \sigma_{q_1}), \ (p\langle i \rangle, \sigma_1), \ \dots \\ t_2 &= \dots \ (q_2\langle j_2 \rangle, \sigma_{q_2}), \ (p\langle i \rangle, \sigma_2), \ \dots \end{aligned} \quad (5)$$

Because $(F_{\text{Uni}}^{\#} \ uni) \ p \ x = \mathbf{true}$ by assumption, it follows from Definition 4.7 that $f_{qp}^{\#} (uni \ q) \ x = \mathbf{true}$ for each predecessor q of p and that $H_p = \mathbf{true}$.

By the latter and by Definition 4.3 and Lemma 4.6, we have that $q_1 = q_2$ and $j_1 = j_2$. Because $q_1 = q_2$, we have $\sigma_1 = f_{q_1 p} \ \sigma_{q_1}$ and $\sigma_2 = f_{q_1 p} \ \sigma_{q_2}$, i.e. both edge transformers are the same. Because $j_1 = j_2$, there is a set $S' \in \gamma_{\text{UniS}} (uni \ q)$ such that $\{\sigma_{q_1}, \sigma_{q_2}\} \subseteq S'$ and therefore there is a set

$$S \subseteq (\mathfrak{f}_{q_1 p} \circ \gamma_{\text{UniS}}) (uni \ q_1)$$

with $\{\sigma_1, \sigma_2\} \subseteq S$. By the fact that $f_{q_1 p}^{\#} (uni \ q_1) \ x = \mathbf{true}$ and the soundness of the edge transformers (4) it follows that $\sigma_1 \ x = \sigma_2 \ x$. \square

4.4 Outline of the Soundness Proof

As discussed in Section 4.2 we want to prove an equivalence between splits_T and splits . The proof for the inclusion $\text{splits}_T \ p \subseteq \text{splits} \ p$ works by transforming two (instance-)disjoint paths in the tagged CFG G_T into two (node-)disjoint paths in the head-rewired CFG (cf. Section 2.3).

Reconsider Equation (3) where r_1 and r_2 are disjoint. By our normalizations on the CFG (cf. Section 3.1), q_1 and q_2 have the same innermost loop. The first observation is that there is no loop that contains q_1 but not the split node s (cf. Lemma 4.9). This allows for inductively constructing disjoint paths on the head-rewired CFG by exiting the loops that contain s one by one. Formally, the main proof is carried out by induction on $\text{depth} \ s - \text{depth} \ q_1$ (Lemma 4.19).

In the base case, the tags are equal thus the tagged paths are already *node-disjoint* (cf. the first part of Lemma 4.13). The disjoint paths on the head-rewired CFG are exactly the underlying CFG paths, except that visited inner loops are immediately exited using the newly introduced exit edges on the head-rewired CFG (cf. Lemma 4.16).

If the split node s is inside of a loop h that does not contain q_1 (and let h be the outermost such loop), only one of the tagged paths—w.l.o.g. r_2 —can take a back edge of h (Lemma 4.10). Otherwise the same instance of h would be visited on both.

Similarly to the base case, the prefix of r_2 up to (including) h is node-disjoint to the other tagged path (cf. the latter part of Lemma 4.13)¹³ and we can re-connect this prefix to the suffix of r_2 that starts at the exit of h . The suffices of r_1 and r_2 from the exits of h yield disjoint paths on the head-rewired CFG in the same way as it is shown in the base case. Lemma 4.13 is formulated in a more general way to capture both cases.

The inductive proof is not in the final soundness theorem (Theorem 4.20) but instead in a lemma that argues about inhomogeneous loop exits (Lemma 4.19). An inductive proof of the proposition of Theorem 4.20 would yield an unusable induction hypothesis, because of the actual join of the tagged paths at p . Thus joining tagged paths from inhomogeneous loop exits is handled in Theorem 4.20.

4.5 Relating Tagged Paths and Paths in the CFG

This subsection investigates the relation between tagged paths and paths in the CFG. The main result is Lemma 4.13 which is useful to derive node-disjoint paths from instant-disjoint paths.

We generalize the setting from Equation (3) by relaxing the requirement of equal target instances to equal target tags:

$$\begin{array}{c}
 s\langle k \rangle \rightarrow_T \underbrace{u_1\langle l_1 \rangle \rightarrow_T^* \quad q_1\langle j_1 \rangle}_{r_1} \rightarrow_T p_1\langle i \rangle \\
 s\langle k \rangle \rightarrow_T \underbrace{u_1\langle l_2 \rangle \rightarrow_T^* \quad q_1\langle j_2 \rangle}_{r_2} \rightarrow_T p_2\langle i \rangle.
 \end{array} \tag{6}$$

Let r_1 and r_2 be disjoint paths on G_T . Here, we additionally require q_1 and q_2 to be in the same loop, i.e. $h_{q_1} = h_{q_2} =: h_q$ (which is guaranteed by the CFG normalizations in Equation (3)). W.l.o.g. we assume $j_1 \leq j_2$. First, we prove some consequences of this setup.

✦ **LEMMA 4.9.** $s \in h_q$.

PROOF. Otherwise h_q would occur in both r_1 and r_2 with the same tag. □

The next lemma states the absence of back edges of the loop h_q in r_1 .

✦ **LEMMA 4.10.** *If $h_q\langle m \rangle \in r_1$ then there is a tag m' such that $m = m'0$ (i.e. the loop h_q is entered).*

PROOF. Assume the contrary, h_q is visited by a back edge. Since every back edge increases the tag and because we have $j_1 \leq j_2$, the other path r_2 must contain the same instance of h_q . □

✦ **LEMMA 4.11.** $r_1 \cup r_2 \subseteq h_q$.

PROOF. Because of $s \in h_q$ a node outside of h_q would require a loop exit of h_q . If there is only an exit on one of the tagged paths r_1 and r_2 , then the tag cannot reconverge at p_1 and p_2 . In the other case both r_1 and r_2 have to contain the same instance of the entry of h_q . □

¹³If h is not the innermost loop of s , we additionally use the induction hypothesis for the inner loops.

We state the main result of this subsection (cf. Lemma 4.13) in a more general setting than the one in Equation (6). This is necessary to make use of it in different ways in Section 4.6. The next lemma shows that Equation (6) indeed implies the requirements of Lemma 4.13.

✦ **LEMMA 4.12.** *If $h_s = h_q$ then $tl\ j_1 = tl\ j_2$ (i.e. j_1 and j_2 differ at most at the last index),*

$$l_1 = j_1 \vee (l_1 = j_1 0 \wedge u_1 \text{ is a header})$$

$$\text{and } l_2 = j_1 \vee (l_2 = j_1 0 \wedge u_2 \text{ is a header}) \vee u_2 = h_q.$$

PROOF. Lemma 4.11 prohibits back edges of loops outside of h_q . Consequently, we have $k = j_1$ and only the latest part of the tag can differ at q_1 and q_2 , thus $tl\ j_1 = tl\ j_2$. Because of $h_s = h_q$ and Lemma 4.11 neither $s \rightarrow_T u_1$ nor $s \rightarrow_T u_2$ can be an exit edge. Lemma 4.10 gives us that $s \rightarrow_T u_1$ is not a back edge and if $s \rightarrow_T u_2$ is a back edge, it is a back edge of h_q . By Definition 3.1 only the stated values for the tags are possible. \square

Now we leave the setting of Equation (6) and show the general lemma to translate disjoint G_T paths to disjoint paths on the CFG.

✦ **LEMMA 4.13.** *For nodes u_1, u_2, q_1, q_2 assume $h_{q_1} = h_{q_2} =: h_q$ and $u_1, u_2 \in h_q$. Let l_1, l_2, j_1, j_2 be tags such that $tl\ j_1 = tl\ j_2$,*

$$l_1 = j_1 \vee (l_1 = j_1 0 \wedge u_1 \text{ is a header})$$

$$\text{and } l_2 = j_1 \vee (l_2 = j_1 0 \wedge u_2 \text{ is a header}) \vee u_2 = h_q$$

hold. Let the paths

$$r_1 := u_1 \langle l_1 \rangle \rightarrow_T^* q_1 \langle j_1 \rangle$$

$$r_2 := u_2 \langle l_2 \rangle \rightarrow_T^* q_2 \langle j_2 \rangle$$

be disjoint. If $j_1 = j_2$, then r_1 and r_2 are node-disjoint \spadesuit . Otherwise there is a prefix r'_2 of r_2 that ends in h_q and r and r'_2 are node-disjoint \spadesuit .

PROOF SKETCH. First note that Lemma 4.10 and Lemma 4.11 also hold in this setting (with analogous proofs). We investigate two different cases:

$j_1 = j_2$: In this case r_1 and r_2 are interchangeable, thus r_2 also does not contain back edges of h_q .

The tag semantics and the properties shown above ensure that, except for inner loops, every instance in r_1 and r_2 has the tag j_1 . But no inner loop h' can be visited on both r_1 and r_2 because they would have the same instance on the entry. (In particular this holds for u_1 and u_2 .) Thus instance-disjointedness implies node-disjointedness.

$j_1 < j_2$: Let r'_2 be the prefix of r_2 such that the last member of r'_2 is the first occurrence of h_q in r_2 . (Non-existence would imply $j_1 = j_2$.) Analogous to the first case. \square

4.6 Syntactical Characterisation of Inhomogeneity

In this subsection we construct a *finite DAG* on which we define the set *splits* similarly to Definition 4.5. With the use of Lemma 4.13 we prove the equivalence between both sets, yielding an efficient way to compute witnesses of local inhomogeneity.

✦ **DEFINITION 4.14** (HEAD-REWired CFG). *The head-rewired CFG is the graph $G_\star := (V, \rightarrow_\star, rt)$ where*

$$\rightarrow_\star \triangleq \rightarrow_G \setminus \{h \rightarrow v \mid h \text{ is a header}\}$$

$$\cup \{h \rightarrow e \mid e \text{ is an exit of loop } h\}.$$

The new edges are called *early-exits*.

✦ **LEMMA 4.15 (EXPANSION).** *Let $\pi := p \rightarrow_{G_\star}^* q$. Then there is a path $\phi \supseteq \pi$ such that $\phi = p \rightarrow^* q$.*

PROOF. For any edge $p \rightarrow_{G_\star} q$: If it is an exit between a header and an exit, then take some path through the loop p in G . Such a path exists because every exit node is reachable from its corresponding header. In any other case take the same edge in G as in G_\star . \square

✦ **LEMMA 4.16 (CONTRACTION).** *Let $\pi := p \rightarrow^* q$ be a path. If $p \in h_q$ and either $h_q \notin \pi$ or $q = h_q$, then there is π' such that $p \rightarrow_{G_\star}^* q$ and $\pi' \subseteq \pi$.*

PROOF. By induction on the loop depth between p and q . If they are in the same loop the outgoing edges from h_q are not used in π and any other edge from G also exists in G_\star . For every loop h strictly inside of h_q : If there is a back edge of h in π the back edge is also traversed on π' and afterwards the exit of that loop in π is taken directly from h . This process builds up a path π' that is a subset of π . \square

✦ **LEMMA 4.17.** *G_\star is a finite DAG.*

PROOF. G_\star is finite since G is finite. By expansion the reachability relation on G_\star is included in the one on G , thus every loop on G_\star is also a loop on G . Let h be a loop header. The rewiring of outgoing edges of h to its exits renders any other node *inside* of h unreachable from h . Thus every loop of G is cut and ultimately G_\star is acyclic. \square

✦ **DEFINITION 4.18.**

splits $p \triangleq \{s \in V \mid \exists \pi, \phi, \text{ that are disjoint paths from the } G \text{ successors of } s \text{ to } p \text{ in } G_\star\}$.

The following lemma contains the critical proof step towards $\text{splits}_T p \subseteq \text{splits } p$: The node-disjoint paths *inside* a loop (as given by Lemma 4.13) are connected to the head-rewired CFG by replacing reiterations of loops with early exits. This process is done by induction on the loop depth.

✦ **LEMMA 4.19 (INHOMOGENEOUS LOOP EXITS).** *For $n_1, n_2 \in \mathbb{N}$, consider the paths*

$$\begin{aligned} s\langle k \rangle &\rightarrow_T u_1\langle l_1 \rangle \xrightarrow{r_1}^* q_1\langle i n_1 \rangle \rightarrow_T e_1\langle i \rangle \\ s\langle k \rangle &\rightarrow_T u_2\langle l_2 \rangle \xrightarrow{r_2}^* q_2\langle i n_2 \rangle \rightarrow_T e_2\langle i \rangle, \end{aligned} \quad (7)$$

and assume $h_{q_1} = h_{q_2} =: h_q$ and that r_1 and r_2 are disjoint. Then there are nodes q'_1, q'_2 fulfilling the path conditions

$$\begin{aligned} s &\rightarrow \underbrace{u_1 \xrightarrow{r'_1}^* q'_1}_{r'_1} \rightarrow_{G_\star} e_1 \\ s &\rightarrow \underbrace{u_2 \xrightarrow{r'_2}^* q'_2}_{r'_2} \rightarrow_{G_\star} e_2, \end{aligned} \quad (8)$$

and such that r'_1 and r'_2 are disjoint.

PROOF SKETCH. W.l.o.g. assume $n_1 \leq n_2$. Because $s \in h_q$ (cf. Lemma 4.9) we have $\text{depth } s \geq \text{depth } q_1$. This allows for a proof by induction on $d := \text{depth } s - \text{depth } q_1$.

$d = 0$ ($\Leftrightarrow \text{depth } s = \text{depth } q_1$): Because of Lemma 4.12 we can apply Lemma 4.13, this gives two subcases (note that $n_1 = n_2 \Leftrightarrow j_1 n_1 = j_1 n_2$):

If $n_1 \neq n_2$ there is a prefix r'_2 of r_2 ending in h_q such that r_1 and r'_2 are node-disjoint. Contracting the CFG paths r_1 and r'_2 using Lemma 4.16 we get two disjoint paths in G_\star . By the definition of G_\star (Definition 4.14) there is an edge from h_q to e_2 .

If $n_1 = n_2$ then r_1 and r_2 are disjoint. Again, with Lemma 4.16 we get disjoint paths in G_\star .

$\mathbf{d} \rightarrow \mathbf{d} + 1$ ($\Leftrightarrow \text{depth } s > \text{depth } q_1$): Let h' be the header of the outermost loop containing s that is still strictly inside h_q .¹⁴ Let e'_1 and e'_2 be the first exits of h' on r_1 and r_2 , respectively. The way h' is chosen gives $\text{depth } h' = \text{depth } q_1 + 1$ thus $\text{depth } s - \text{depth } h' = d$. This allows for the application of the induction hypothesis to get disjoint G_\star paths r'_1 and r'_2 from s to e'_1 and e'_2 , respectively. Now, e'_1 and e'_2 are in the same loop as q_1 and q_2 . The prerequisites of Lemma 4.13 can be shown similarly as in Lemma 4.12.¹⁵ By the same argument as in the base case, applying Lemma 4.13 delivers disjoint G_\star paths from e'_1 to e_1 and from e'_2 to e_2 . Since these paths are completely outside of the loop h' , they are disjoint to the paths given by the induction hypothesis, thus the concatenation of these paths finishes the proof. \square

Lemma 4.19 delivers disjoint G_\star paths ending in loop exits. For the soundness proof, this covers the case where tagged paths join at loop exits as in Figure 1b (but also for nested loops). We argue similarly in the case of joins of tagged paths that are joins on the CFG as well (cf. Figure 1a). The general case, visualized in Figure 1c, is shown by combining both concepts.

\heartsuit **THEOREM 4.20** (SOUNDNESS OF *splits*).

$$\text{splits}_T p \subseteq \text{splits } p$$

PROOF. Let $s \in \text{splits}_T p$. Then there is a tag k and disjoint paths

$$\begin{aligned} r_1 &:= u_1 \langle l_1 \rangle \rightarrow_T^* q_1 \langle j_1 \rangle \\ r_2 &:= u_2 \langle l_2 \rangle \rightarrow_T^* q_2 \langle j_2 \rangle \end{aligned}$$

originating in $s \langle k \rangle$ and rejoining in $p \langle i \rangle$. Because of Requirement 1 (Section 3.1) and the definition of the tag semantics either $q_1 \rightarrow p$ and $q_2 \rightarrow p$ are both loop exit edges or neither is. In the first case we can simply apply Lemma 4.19.

In the case where neither edge is an exit edge, the tag semantics give $j_1 = j_2$. We distinguish two different cases:

$\mathbf{p} \in \mathbf{h}_s$: Similarly to the base case of Lemma 4.19, we directly apply Lemma 4.13. Since we have $j_1 = j_2$, we do not have to consider the other sub-case of Lemma 4.13. With Lemma 4.16 we get the disjoint paths in G_\star .

$\mathbf{p} \notin \mathbf{h}_s$: Let h' be the outermost loop such that $s \in h'$ and $p \notin h'$. Analogously to the proof of the step case in Lemma 4.19. \square

In the completeness proof we make use of the fact that every node is preceded by all its loop headers with a compatible tag. This is a consequence of dominance of loop headers on reducible CFGs.

\heartsuit **LEMMA 4.21** (PRECEDENCE ON G_T). *Let $\pi := rt \langle \rangle \rightarrow_T^* p \langle i \rangle$ be the G_T path to the instance $p \langle i \rangle$. Then we have for every header h with $p \in h$ that $h \langle i' \rangle$ precedes $p \langle i \rangle$ on π for some prefix i' of i with $|i'| = \text{depth } h$.*

We want to show that the use of the head-rewired CFG does not overestimate splits_T . The idea of the proof is to construct the paths on the G_T from the paths on G_\star . This is done by replacing every early-exit edge with some path through the corresponding loop while every other edge can

¹⁴I.e. $h' \in h_q \wedge h' \neq h_q$

¹⁵Its proof can be modified to the case where u_1 and u_2 are not required to succeed s , if we instead require $h_q = h_{u_1} = h_{u_2}$.

remain the same. We then only have to argue that the *instances* on the newly introduced passes through loops are disjoint to the respective other path.

THEOREM 4.22 (COMPLETENESS OF *splits*).

$$\mathit{splits} p \subseteq \mathit{splits}_T p$$

PROOF. Let $s \in \mathit{splits} p$. Then there are successors u_1, u_2 of s in G and disjoint paths $r_1 := u_1 \rightarrow_{G_\star}^* q_1$ and $r_2 := u_2 \rightarrow_{G_\star}^* q_2$ with $q_1 \rightarrow_{G_\star} p$ and $q_2 \rightarrow_{G_\star} p$.

Let k be the tag consisting of *depth* s zeros, i.e. $k := 0^{\mathit{depth} s}$. The choice of k guarantees that there are no back edges on the path $rt\langle \rangle \rightarrow_T^* s\langle k \rangle$. Let r'_1 and r'_2 be the expanded CFG paths of r_1 and r_2 (cf. Lemma 4.15). r'_1 and r'_2 are disjoint: For any non-early-exit edge (i.e. an edge that exists in the CFG) exactly the same nodes are used as in the original edge. Let h be the source of an early-exit edge and let j be the corresponding tag. W.l.o.g. we have $h \in r_1$ and $h \notin r_2$. Assume there is an instance $x \in r_2$ in the loop h whose tag is prefixed by j . By Lemma 4.21 there is an instance of h with tag j in the tagged path to x . Because $h \notin r_2$, h is on the path to $s\langle k \rangle$. Since there are no back edges on the path to $s\langle k \rangle$ we have $j = 0^{\mathit{depth} h}$. But then there are two occurrences of $h\langle j \rangle$ on the path $rt\langle \rangle \rightarrow_T^* u_1\langle j_1 \rangle \rightarrow_T^* q_1\langle j_1 \rangle$ which contradicts acyclicity (cf. Lemma 3.4). \square

5 EFFICIENTLY COMPUTING ENDS OF DISJOINT PATHS IN A DAG

Because of Theorem 4.20 every join node for which a given split node s is relevant is characterized by two disjoint paths in G_\star that originate from two successors of s (in G) to two predecessors of that join node. Therefore, we now present a simple and efficient algorithm that computes *all* join points of pairs of disjoint paths that originate from a given node set S (here the successors of node s).

Assume we are given a set S of source nodes in a DAG for which we want to be able to compute information such that, given two nodes p and q , we can answer the question “Are p and q end nodes of two node-disjoint paths that originate from two nodes in S ?”

Consider a node p that is reachable from some node in S and consider the set of all paths

$$P_p \triangleq \{\pi \in s \rightarrow^* p \mid s \in S\}$$

from nodes in the source set to some node p . The set

$$D_p \triangleq \{d \mid \forall \pi \in P_p. d \in \pi\}$$

contains all nodes in the DAG G that lie on *every* path in P_p . (The nodes in D_p have similar characteristics as the dominators of p . We discuss this relationship briefly at the end of this section.)

The set D_p is not empty (it contains at least p itself) and all nodes in D_p are totally ordered with respect to the reachability relation of the DAG. So there is a minimum (earliest) node p^* in D_p from which all other nodes in D_p are reachable.

Nodes like p^* are interesting because a pair of them guarantee the existence of disjoint paths: Assume in the following that we are given two different such nodes p^* and q^* .

✦ **LEMMA 5.1.** *There exist two different nodes s_1 and s_2 in S that are the origins of two disjoint paths to p^* and q^* .*

PROOF. Because $p^* \neq q^*$, there is no node that lies on all paths from S to p^* and from S to q^* . Therefore, a minimal separator¹⁶ between S and $\{p^*, q^*\}$ contains at least two nodes. By the directed node-disjoint version of Menger’s theorem [e.g. Schrijver 2017, Theorem 3, Chapter 3], there exist two nodes $\{s_1, s_2\} \subseteq S$ such that there are two node-disjoint paths from s_1 to p^* and from s_2 to q^* . \square

¹⁶a set of nodes whose removal will disconnect two given sets of nodes.

♣ **LEMMA 5.2.** *There exist two disjoint paths $p^* \rightarrow^* p$ and $q^* \rightarrow^* q$.*

PROOF. By contradiction: Consider two paths $p^* \rightarrow^* p$ and $q^* \rightarrow^* q$ and assume they have a node in common. Let x be the first node that both paths have in common. There are two cases: (1) $x = p^*$ or $x = q^*$: Assume w.l.o.g. $x = p^*$. Because $p^* \neq q^*$ by assumption, p^* is reachable from q^* . Because of Lemma 5.1 there are two disjoint paths to p^* and q^* . Hence there is a path from S to p^* to q that does not contain q^* which violates the definition of q^* . (2) $x \neq p^*$ and $x \neq q^*$: Combining the one path $p^* \rightarrow^* x$ with the suffix $x \rightarrow^* q$ of the other path yields a path to q that does not contain q^* which contradicts its definition. \square

♣ **LEMMA 5.3.** *If $p^* \neq q^*$ then there exist two different nodes s_1 and s_2 in S and two disjoint paths from s_1 to p and from s_2 to q .*

PROOF. Combining Lemma 5.1 and Lemma 5.2. \square

```
def labelDisj(G: DAG, sources: Set[Node]):
  # label each source with itself
  labels = { s:s for s in sources }
  for p in G.toposort_of_nodes():
    # obtain all preds reachable from sources
    reach = filter(lambda p: p in labels.keys(), p.preds())
    if len(reach) > 0:
      # if this node is reachable from S
      plabs = map(lambda p: labels[p], reach)
      if len(set(plabs)) == 1:
        # if there is at least one predecessor
        # and all predecessors have identical labels
        labels[p] = plabs[0]
      else:
        labels[p] = p
  return labels
```

Algorithm 1. Computing ends of disjoint paths in DAGs

Algorithm 1 exploits Lemma 5.3 by computing for each node p the corresponding node p^* by the following observation: If for two predecessors p_1 and p_2 of p that are reachable from S it holds that $p_1^* \neq p_2^*$, then by Lemma 5.3 there are two disjoint paths from S to p_1 and p_2 that join at p and $p^* = p$. Otherwise, p^* is the same node for all reachable predecessors, we have $p^* = p_1^*$. Algorithm 1 applies this local consideration along a topological sort of the input DAG.

LEMMA 5.4. *The following is a loop invariant of Algorithm 1:*

$$\forall p. (\exists x. \text{labels}(p) = x) \iff \text{reachable}_S(p)$$

PROOF. Straightforward. \square

♣ **LEMMA 5.5.** *The following is a loop invariant of Algorithm 1:*

$$\forall p. (\exists x. \text{labels}(p) = x) \implies \text{labels}(p) = p^*$$

PROOF. Because of Lemma 5.4 all reachable predecessors of p are labeled. Because of the precondition for each predecessor p of p that is reachable from S it holds that $\text{labels}(p) = p^*$. If for all (reachable from S) predecessors p_1, \dots, p_n it holds that $p_1^* = \dots = p_n^*$, then p_1^* also is on every path from S to p and therefore $p^* = p_1^*$ (♣). If there are $i \neq j$ such that $p_i^* \neq p_j^*$, then $p^* = p$ (♣). \square

THEOREM 5.6. *If two nodes have different labels after running Algorithm 1, they are end points of two disjoint paths originating in S .*

PROOF. Directly from Lemma 5.3 and Lemma 5.5. \square

COROLLARY 1. *If Algorithm 1 labels a node $p \notin S$ with itself, then p is the join point of two disjoint paths starting in S .*

Complexity. Topological sorting takes $O(|V| + |E|)$ time. The algorithm itself runs in $O(|V| + |E|)$ as well. After the algorithm labelled the graph for a set of source nodes, the question if there are disjoint paths from S to any two nodes is then answered in $O(1)$ by comparing their labels.

A note on (post) dominance frontiers. In a lot of related work, iterated dominance or post-dominance frontiers (IDFs) are used to compute disjoint paths. However, dominance frontiers over-approximate the set of join points we are interested in because they implicitly consider joins with the root (end) of the CFG. The relation between IDFs and join points that is proven by Cytron et al. is $IDF^+(S) = Joins(S \cup \{rt\})$ [Cytron et al. 1991]. For our setting, using IDFs would cause imprecision because more nodes might be marked divergent than necessary.

6 DEFINITE REACHING DEFINITIONS ANALYSIS

Reconsider Figure 1a and assume we are interested in the uniformity value of z at program point P . Assume that c is not uniform at S . Then, $F_{Uni}^\#$ produces an abstract state in which z is varying although z clearly is uniform. This is because S is a relevant split for P with a non-uniform branch predicate: if a set of traces is not locally homogeneous, the transformer yields **false** although all traces may still agree on the value of z because z was defined at an instance that all these traces include (like the instance $S\langle \rangle$ in this example).

To remedy this problem, we adapt a classic reaching definitions data-flow analysis in the following way. First, one says that a definition¹⁷ q of a variable x *reaches* p along a path $\pi : q \rightarrow p$, if π does not contain a further definition of x . Classic reaching definitions analysis (see for example [Nielson et al. 1999]) collects *potential* reaching definitions, i.e. computes the union of all possible definitions that reach p . In the example, we discussed above however, we are interested in associating a program point p with a single definition that reaches p *definitely*, i.e. along all paths to p or phrased in different terms, a reaching definition that *dominates* p . So our analysis is a *definite* reaching analysis which makes it a hybrid of the classic reaching definition analysis by exchanging set union by set intersection in the abstract transformer (Definition 6.4) and a data-flow formulation of dominance (e.g., see [Cooper et al. 2001]).

We extend the classic data-flow formulation to an abstract interpretation by equipping it with a concretization that delivers the invariants necessary to solve the problem discussed at the beginning of this section.

✦ **DEFINITION 6.1** (DOMINANCE DOMAIN).

$$RDS \triangleq \text{Var} \rightarrow \mathcal{P}(\text{Lab}) \quad RD \triangleq \text{Lab} \rightarrow RDS$$

Lifting subset inclusion to **RD** in the standard way makes **RD** a complete lattice. The concretization says that a variable x is *unchanged* at a program point p with respect to a program point q if on every trace that contains an instance $p\langle i \rangle$ of p there is an instance $q\langle j \rangle$ of q that *precedes* $p\langle i \rangle$ and the value of x in the configurations that correspond to the two instances are equal.

✦ **DEFINITION 6.2** (REACHING DEFINITIONS CONCRETIZATION).

$$\begin{aligned} \gamma_{RD} : RD &\rightarrow \mathcal{P}(\text{Tr}) \\ rd &\mapsto \{t \mid \forall p q i x \sigma. \\ &\quad q \in rd p x \Rightarrow (p\langle i \rangle, \sigma) \in t \Rightarrow \\ &\quad \exists j \sigma'. (q\langle j \rangle, \sigma') \in t \wedge q\langle j \rangle \preceq p\langle i \rangle \wedge \sigma x = \sigma' x\} \end{aligned}$$

¹⁷i.e. the program point where x is defined

✦ **THEOREM 6.3.** γ_{RD} is meet-preserving.

✦ **DEFINITION 6.4** (REACHING DEFINITIONS ABSTRACT TRANSFORMER).

$$\begin{aligned} f_{qp}^\# : RDS &\rightarrow RDS \\ c &\mapsto \lambda x. \begin{cases} \emptyset & \text{isdef } q p x \\ c x & \text{otherwise} \end{cases} \\ F_{RD}^\# : RD &\rightarrow RD \\ rd &\mapsto \lambda p. \lambda x. \{p\} \cup \bigcap_{q \rightarrow p} f_{qp}^\#(rd q) x \end{aligned}$$

✦ **THEOREM 6.5.** $F_{RD}^\#$ is sound.

We exploit this analysis to take the uniformity information of a definitely reaching definition into account. Coming back to Figure 1a, we can prove z uniform at P because it is *unchanged* with respect to program point S and z is uniform there.

There is one caveat though. Suppose some variable's x definition at q definitely reaches p , and q is nested in a loop not containing p . Then, we cannot conclude that x is uniform at p even if it is uniform at q . For an example, see Figure 1b. Variable x is uniform at C and C definitely reaches P but x is *not* uniform at P . This is caused by the divergence at the exit of the loop. Hence, we can only use the “definitely reaches” relationship between two program points p and q to transfer uniformity information from q to p if q precedes p homogeneously in every trace, i.e. if $q\langle j \rangle < p\langle i \rangle$ in some trace t and $q\langle j' \rangle < p\langle i \rangle$ in t' , then $j = j'$.

Considering the tag semantics (Definition 3.1) it can be seen that as long as there is no loop that contains q and does not contain p , q always precedes p homogeneously. If this is not the case, an additional condition needs to make sure that all loops surrounding q are exited homogeneously. This is captured in the following more precise version of the uniformity transformer that uses the results of the definitely reaching definitions analysis.

✦ **DEFINITION 6.6** (UNIFORMITY TRANSFORMER v2).

$$\begin{aligned} F_{\text{Uni},2}^\# \text{ uni} &\triangleq \lambda p. \lambda x. (F_{\text{Uni}}^\# \text{ uni}) p x \\ &\vee \bigvee_{q \in (rd p x) \setminus \{p\}} \left(\text{uni } q x \wedge \bigwedge_{s \in \text{lsplits } q} \text{uni } s x \right) \quad \text{with } \text{lsplits } q \triangleq \bigcup_{\substack{e \text{ is loop exit of a loop containing } q \\ e \rightarrow^* p \text{ that does not contain a back edge}}} \text{splits } e \end{aligned}$$

✦ **THEOREM 6.7.** $F_{\text{Uni},2}^\#$ is sound.

6.1 A Note on SSA

One of the major motivations for SSA is to enable sparse analyses by unifying the notion of a variable and a program point. This works if there is a new SSA value for each program point where a variable can change its abstract value. In this way ϕ -functions relate to the program points where data flow values are joined.

For the divergence analysis discussed in this and the last chapter this also works with one exception: Variables can change their uniformity at loop exits. Loop-closed SSA (LC-SSA, see e.g. [Latner 2004; Pop et al. 2009]) solves this issue by inserting additional ϕ -functions at loop exiting nodes for variables that are modified in a loop and live beyond its exit. These extra LC-SSA ϕ -functions can carry the data flow information for the corresponding non-SSA variables at loop exits. Furthermore, LC-SSA breaks the live ranges of variables that are defined in and live out

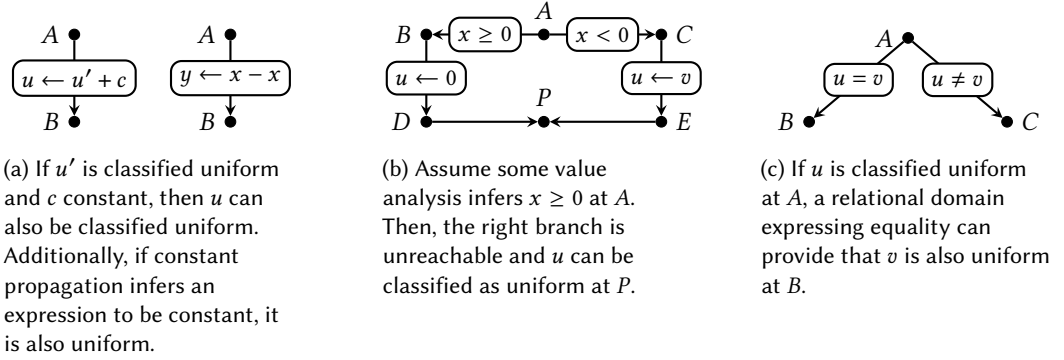


Fig. 3. Examples for cooperation with value analyses.

of a loop. Therefore, on LC-SSA $F_{\text{Uni},2}^\#$ and $F_{\text{Uni}}^\#$ give the same results and running the definitely reaching definitions analysis is not necessary.

We leave a formal treatment of an SSA version of our analysis for future work.

6.2 Combining Uniformity with Other Analyses

The divergence analysis presented in this paper can be combined with other analyses using reduced products [Cousot and Cousot 1979], a standard abstract interpretation technique that allows analyses to share their information. Essentially, the concretization of a product of two analyses is the intersection of the two individual concretizations. To make this work in our setting, the concretization $\gamma_{\text{D}} : \mathbb{D} \rightarrow \text{Tr}$ of a “non-hyper” value analysis D needs to be intersected with each set of traces in γ_{Uni} :

$$\gamma_{\text{Uni} \times \text{D}}(\text{uni}, d) \triangleq \{T \cap \gamma_{\text{D}} d \mid T \in \gamma_{\text{Uni}} \text{uni}\}$$

We also combine **Uni** and **Unch** in this way in the Coq development that accompanies this paper to prove Theorem 6.7.

Most importantly, constant values are uniform: This means that the uniformity edge transformers that compute the abstract information of the effects on edges (such as assignments) can easily take information from constant propagation into account. But also other analyses can provide helpful information to our analysis. Figure 3 gives some examples for integrating uniformity analyses with other value analyses.

- Adding a constant to a uniform value yields a uniform value (Figure 3a).
- Constant branch predicates lead to unreachable code (Figure 3b)
- Proving equality of a varying variable to a uniform variable makes that variable uniform (Figure 3c).

7 EXPERIMENTAL EVALUATION

We evaluate our new analysis algorithm with the Region Vectorizer (RV) [Moll and Hack 2018]. RV is a vectorizer for SPMD programs and operates on LLVM IR [Lattner and Adve 2004], which is an SSA-based intermediate representation. Our implementation follows the LC-SSA scheme laid out at the end of Section 6. RV uses the results of a divergence analysis to retain scalar computations and branches in the vectorization of CFGs. Therefore, the precision of the divergence analysis influences the quality of the vectorized code.

7.1 Baselines

We compare our new algorithm against two other divergence analyses. The first analysis is the commonly used divergence analysis that ships with LLVM (basically an instance of [Coutinho et al. 2011]). As discussed in Section 8, this analysis is unsound on general reducible CFGs and requires stronger syntactical restrictions (see Section 8). We will refer to this analysis as *llvm*. Since *llvm* is unsound on general reducible CFGs, we also compare against a state-of-the-art sound analysis that supports at least reducible CFGs, which is the information-flow analysis by Wasserrab et al. [2009]. This analysis is based on slicing (i.e. control dependence) to identify the relevant splits, similar to the approaches discussed in the section *control dependence* in Section 8. The notion of control dependence in this algorithm is based on strong post dominance to account for the divergence caused by divergent loop exits that is missed if regular post dominance is used. We compute the strong post-dominance frontier (short *spd*) and transfer the algorithm to SSA form programs as follows: If a block is a loop exit, it is divergent if it lies in the transitive closure of the *spd* frontier of a divergent branch. Otherwise, a join block node is divergent if at least one predecessors of the block is in the transitive closure of the *spd* frontier of a divergent branch. We call this setup *strongpd*.

We refer to the analysis presented in this paper simply as the *new* configuration.

Analysis runtime (i.e. compile time) is measured on an AVX2 machine (Intel(R) Core(TM) i7-8565U CPU) with hyper-threading and turbo boost disabled. The reported analysis runtimes are geometric means of 30 full divergence analysis runs.

7.2 Benchmark Kernels

We evaluate each of the three divergence analyses in two disciplines. First, on four different OpenCL benchmark suites to obtain an insight into the analysis runtime on kernels with structured control flow. For the second part, we consider a set of SPMD kernels that heavily rely on a vectorizer that is capable to exploit uniformity information to achieve good performance. These SPMD benchmarks feature complex, unstructured, control-flow and put more stress on the divergence analysis than the structured OpenCL kernels. Figure 4 shows the benchmark results aggregated per benchmark suite. Appendix A has the detailed results on a per-kernel basis.

OpenCL. We extract the kernels with the POCL OpenCL driver Jääskeläinen et al. [2015] and apply each divergence analysis (*new*, *llvm* and *strongpd*) to them. The benchmarks comprise the OpenCL kernels of SPEC ACCEL [Juckeland et al. 2014], RSBench [Tramm et al. 2014a], XSBench [Tramm et al. 2014b], LuxMark [Bucciarelli et al. 2020] and Hetero-mark [Sun et al. 2016]. The results, shown in Figure 4a, reflect how the analysis algorithm performs on these kernels.

SPMD-aware Traversal Codes. To put more stress on the divergence analysis, we consider two benchmarks that were designed explicitly with SPMD code generation in mind. Rodent [Pérard-Gayot et al. 2019] is a state-of-the-art ray tracing framework whose SPMD code path heavily relies on RV for vectorization. We run Rodent in the traversal benchmarking mode on the *Sponza* scene. Rodent traverses a BVH with a speculative stack-based traversal scheme. The traversal kernels are vectorized for packets of rays and single rays.

We further evaluate on a suite of SPMD tree traversal codes that were originally used to evaluate RV [Moll and Hack 2018]. These benchmarks feature more complex control flow than the OpenCL kernels. We again evaluate the analysis runtimes and the quality of results. The results are shown in Figure 4b.

Suite	Statistics vs. <i>strongpd</i>									Analysis time				
	Instructions			Branches			Loops			Control		Full DA		
	Total	Uni	Δ	Total	Uni	Δ	Total	Uni	Δ	$\frac{llvm_{new}}{llvm_{new}}$	$\frac{control_{full_da}}{control_{full_da}}$	μs	$\frac{llvm_{new}}{llvm_{new}}$	$\frac{spd_{new}}{spd_{new}}$
SPEC ACCEL (37)	5734	1695	+278	657	454	+41	110	54	+35	1.13	0.03	379	1.01	1.30
Hetero-Mark (6)	491	118	+26	67	53	+5	16	14	+5	1.04	0.04	292	1.00	1.34
LuxMark (14)	12608	7431	+228	1714	1316	+12	50	8	+8	0.75	0.04	9623	0.98	1.18
Doe Proxy Apps (2)	1573	1409	+66	159	148	+12	15	8	+8	0.97	0.01	4327	1.00	1.30
geom. mean										1.01	0.03		1.00	1.28

(a) OpenCL results.

Suite	Statistics vs. <i>strongpd</i>									Analysis time				
	Instructions			Branches			Loops			Control		Full DA		
	Total	Uni	Δ	Total	Uni	Δ	Total	Uni	Δ	$\frac{llvm_{new}}{llvm_{new}}$	$\frac{control_{full_da}}{control_{full_da}}$	μs	$\frac{llvm_{new}}{llvm_{new}}$	$\frac{spd_{new}}{spd_{new}}$
Rodent (18)	18299	11799	+0	2746	2702	+0	144	144	+0	0.62	0.00	4366	1.00	1.13
Treelogy (13)	2309	1557	+753	402	317	+46	46	36	+14	0.94	0.02	543	1.00	1.29
geom. mean										0.84	0.01		1.00	1.20

(b) SPMD-aware Traversal code results.

Fig. 4. Aggregated benchmark suite results (Number of benchmarks in ‘()’). **Left half:** Entity counts for Instructions, Branches and Loops. For each, we show the total count (Total), number of uniform entities found by the *new* analysis (Uni) and the difference in uniform entities compared to the *strongpd* baseline. **Right half:** Absolute and relative runtime spent in the control-divergence part (i.e. identifying disjoint paths for *new*) of the algorithm. The full data broken down to each individual benchmark can be found in Appendix A.

7.3 Summary of the Results

We call a loop *uniform* if it contains no splits that lead to control-induced divergence of instructions outside the loop. Divergent loops trigger control conversion in the vectorizer, which takes a toll on the efficiency of the generated SPMD code. Therefore, we report the uniformity of loops explicitly in the result tables.

Correctness & Precision. We did not observe irreducible loops (multi-header loops) in the entire set of benchmark suites, hence the *new* analysis was always applicable. The *llvm* analysis delivered the same analysis results as *new*, i.e. the more strict requirements on control flow of the *llvm* algorithm did not show in the kernels. Nevertheless, there are practically-relevant programs that this analysis cannot soundly analyze (see Section 8 and specifically Figure 5b).

new is more precise than *strongpd* in all benchmark suites except Rodent and detects more uniform instructions, branches and loops than *new*. Overall, *new* classifies 5% more instructions as uniform as *strongpd*.

Overall Divergence Analysis Runtime. Comparing *new* to *llvm*, the overall analysis runtime is on par (speedup geomean of 1) with slowdowns of up to 7% for large kernels and speedup of up to 6% on smaller ones of *new*. The *new* analysis is faster than the *strongpd* baseline (speedup geomean of 1.25).

The column “Full DA” includes the startup time of the analysis. For *new* is the time spent to setup the post-order traversal of the head-rewired CFG to compute the toposort. For *strongpd* this is the time needed to construct the (strong) post-dominance frontier. The *llvm* analysis has no startup time as all computations happens on the fly.

The startup time of *strongpd* is higher than that of *new* as the strong post-dominance frontier is computed up front. Out of the 76 evaluated kernels, there are only two cases where the *strongpd* is faster than *new*. The outliers are two kernels with more than 300 branches and divergent control.

For these two, the cost of computing the strong post dominance frontier is amortized by fewer iterations required for the transitive closures (as compared to the *new* analysis which iterates over the control flow edges).

Divergent Branch Analysis Runtime. We now consider the control-dependence part (computing disjoint paths for *new*, iterating the post-dominance frontiers for the others) of each analysis in isolation to better understand the factors that determine how the *llvm*, *strongpd* and *new* analysis perform relative to each other. The *new* divergence analysis spends up to 13% of its runtime computing disjoint paths and only 2% in the geometric mean.

For the OpenCL kernels in the geometric mean, the *new* analysis is slightly faster than the *llvm* analysis (geom mean 1.01). On the SPMD kernels, the *new* analysis is actually slower in the geometric mean (0.84). The results indicate that kernels with fewer branches (roughly up to 24) are faster or at least as fast when analyzed by the *new* analysis compared to *llvm*. This becomes evident by comparing the total number of branches in the OpenCL and SPMD kernels (per kernel results in Appendix A): 20 out of the 31 SPMD kernels, have more than 24 branches. However, only 17 out of the 45 OpenCL kernels have more than 24 branches.

In the worst case, we observe a ratio of 0.45 (`116.histo.intermediates`), meaning that the *new* analysis spends $\times 1.22$ more time in analyzing control divergence than the *llvm* setting. Yet, in the big picture of the full divergence analysis runtime for this kernel, the *new* setting is only about 3% slower. For short control analysis times, the *new* analysis can be much faster as witnessed by some OpenCL kernels. For example, we observe a 60% speedup of the *new* control-divergence analysis over *llvm* for the `122.cfd.time.step` kernel.

8 RELATED WORK

There exists a huge body of related work from various different domains, especially divergence analysis, non-interference analysis, and binding-time analysis. We touch the latter only briefly because it is mostly considered on lambda calculi which is substantially different from a small-step semantics setting like ours. Of the other works we only discuss those that either use abstract interpretation or also target CFGs as their code representation.

Gated SSA. GSA uses γ , μ , and ν “functions” that select SSA values at control flow joins, loop headers, and exits. While ϕ -functions in SSA use control flow to implicitly select one of their operands, their GSA counterparts explicitly use the branch condition to select one of their operands. This makes GSA appealing for divergence analysis because the transformers can now directly use the uniformity of these branch conditions to infer the uniformity of the variables at a join. [Sam-paio et al. \[2013\]](#) exploit this and formulate divergence analysis on GSA and prove it correct with respect to a semantics of that is based on GSA. Therefore, their results depend on the soundness and precision of GSA construction. Hence, we relate the divergence criterion based on disjoint paths that is elaborated in this paper to the existing GSA construction techniques in the following.

Employing the standard GSA construction algorithms ([Ballance et al. \[1990\]](#); [Tu and Padua \[1995\]](#)) and deriving uniformity from the branch predicates in the γ -expressions is less precise than our approach. We demonstrate this on two examples.

Consider [Figure 5a](#). The GSA expression for x at J is $\gamma(p, \gamma(q, \gamma(s, 0, 1), \perp), 0)$. The results from [Section 4](#) state that the uniformity of x at J does not depend on q . Yet, the branch condition q occurs in the GSA expression because the expression includes all control-dependences of the join point, i.e. under which condition the join point is executed, in this example those of J ,

With respect to loops, GSA is less precise because it provides only one loop exit predicate for an entire loop. For example in [Figure 5d](#), the GSA loop exit predicate for loop H is $\gamma(h, 1, \gamma(b, \gamma(c, 1, 0), 0))$. If any of h , b or c are varying, then judging by the γ expression, both

exits C and D are divergent loop exits. By checking for disjoint paths to the latch, our technique infers that uniformity of the exit D is independent of b or c .

Havlak [1993] introduced thinned gated SSA form (TGSA) based on the observation that the γ expressions of GSA contain irrelevant branch conditions. However, the TGSA algorithm is incorrect for loops. In Figure 5d, the TGSA loop predicate for the H loop is $\gamma(b, \gamma(c, true, false), false)$. According to this γ expression the loop will erroneously appear uniform when only h is varying.

Finally, to the best of our knowledge, none of the GSA construction algorithms have been proven correct. Hence, when working with a non-GSA program representation, the correctness proof of Sampaio et al. does not protect against potential flaws of the respective GSA construction algorithm.

Post-Dominator Reconvergence. A large body of earlier work assumes that diverged threads will not reconverge before the immediate post-dominator (IPD) [Coutinho et al. 2011; Farrell and Kieronska 1996; Habermaier and Knapp 2012; Karrenberg 2015]. IPD reconvergence became popular through its adoption in NVIDIA GPUs starting with Tesla [Lindholm et al. 2008]. However, beginning with the more recent Volta GPUs, any two threads may reconverge whenever their program counters agree [NVIDIA 2017, Fig. 21], which may be before the IPD. Similarly, in recent SPMD vectorization techniques [Moll and Hack 2018], reconvergence can happen before the IPD.

When the assumption of IPD reconvergence is lifted, these techniques deliver unsound results on all unstructured CFGs in Figure 5.¹⁸ For example, assuming that p is varying in Figure 5b, the divergence of x at D goes undetected because the immediate post-dominator of P is F .

Chandrasekhar et al. [2019] and Collange [2011] consider all join points above the immediate post-dominator of a divergent branch as joins of disjoint paths. While this repairs the IPD techniques, it is less precise than our approach. Consider the unstructured CFG in Figure 5b. When p is uniform and q is varying, x will be flagged as varying at D despite the fact that q has no influence on the divergence of x .

Alur et al. [2017] formalize divergence analysis on structured syntax but the implementation they use to evaluate their work is based on CFGs. In their implementation, they *always* set the uniformity of ϕ -functions to *varying*. In comparison, the core of our work is to provide a transformer for control-flow joins (which can also be used for ϕ -functions) that provides more precision.

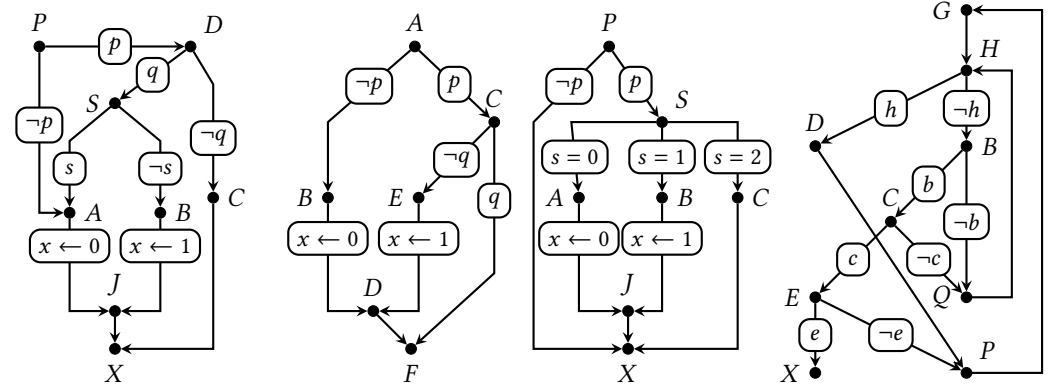
Control Dependence. In these techniques, the (transitive) control dependence of an assignment is used for divergence detection [Lee et al. 2013; Liang et al. 2016]. Figure 5a shows that this is less precise than the analysis presented in this paper. The assignment $x \leftarrow 1$ is control dependent on D . Hence, the uniformity of q will influence the one of x which is not the case in the analysis presented in this paper. Similar observations can be made for all CFGs in Figure 5. One additional complication is that [Lee et al. 2013] does not describe how to handle divergent loop exits.

To improve the precision, some approaches refine control dependences by subtracting the control dependences of the join point from the control dependences of its predecessors. This approach used by the AMD HSAIL¹⁹ driver and recent work on GPU kernel analysis [Lloyd et al. 2019]. However, these approaches are unsound on Figure 5a: B and E are jointly control dependent on A and C . These are also the control dependences of D , subtracting them means that a varying p does not force x to vary at D .

SPMD Languages. SPMD languages like ISPC [Pharr and Mark 2012], Sierra [Leißa et al. 2014], and Hipacc [Reiche et al. 2017] provide abstractions for the SPMD programming model. Their compilers all perform divergence analysis to improve the performance of the generated vector

¹⁸https://bugs.lvm.org/show_bug.cgi?id=42741

¹⁹<https://reviews.lvm.org/D50433#1195085>



(a) Whether x is varying at J depends on p but not on q . Using transitive cdeps includes q spuriously. “Filtering” cdep nodes erroneously discards p .

(b) IPD approaches miss divergence of x at D when p is varying.

(c) Most techniques assume ≤ 2 successors and break when there are more.

(d) B post-dominates H and pre-dominates Q . The TGSSA loop predicate is invalid.

Fig. 5. Examples used in the discussion of related work.

code, however in a syntax-driven way: For example, when there is a divergent branch governing a break in a loop in an ISPC program, ISPC switches to a different code generation scheme for the entire loop, making every variable live out of the loop divergent and guarding all accesses to these variables with masking code. In contrast, our analysis however works directly on reducible CFGs.

Binding-Time Analysis. Aiken and Gay [1998] first noted that binding-time analysis [Jones et al. 1989] as it appears in partial evaluation can detect the uniformity of variables at join points. Binding-time analysis literature (e.g. Consel [1990] or Nielson and Nielson [1988]) almost exclusively deals with functional programs. Among these works, the technique by Auslander et al. [1996] stands out for supporting unstructured control-flow. In their technique, paths formulas are constructed containing only uniform branch conditions. Deciding uniformity then boils down to solving a SAT problem at join points. However, the approach does not treat loops (always flags loop exits as divergent) in contrast to our work.

Non-Interference Analysis. Another (hyper-) property that is very closely related to uniformity and binding-time is *non-interference*. The body of work on non-interference analysis is too big to discuss here in its entirety. We therefore only mention the work that is closely related to ours with respect to abstract interpretation and the way that control flow is handled.

Abadi et al. [1999] first noticed the relation between non-interference and binding-time analysis. Assaf et al. [2017] and Urban and Müller [2018] present abstract interpretations for non-interference concretizing to hyper-semantics which is similar to our work. However, all these approaches use structured control flow induced by inductively-defined syntax. In such a setting, identifying the relevant splits is straightforward because they are contained (as branch expressions) in the syntactic entities (if, while) the abstract transformer operates on. While this makes their analyses simpler, they are hardly applicable in many practical settings where dealing with CFGs is necessary (such as LLVM bitcode or Java byte code).

In a recent publication, [Cousot \[2019\]](#) presents an analysis that is very similar to ours, especially with respect to the flow-sensitivity of uniformity. He too uses inductively defined syntax which simplifies certain aspects of the analysis as laid out above. However, his analysis is more informative in that it does not only classify variables as uniform or varying but reports explicit dependences, i.e. one knows which variables' input values contributed to a variable being varying (dependent) at a certain program point. Furthermore, his semantic definition of dependence (being varying) is different from ours: In our setting, being varying means that there possibly is an *instance* of a program point where two traces disagree on a variable's value. He however records all values a variable assumes at a program point and classifies the variable as dependent if these value lists differ. If one transferred his notion of dependence to the uniform/varying setting, this would lead to more non-uniformity than in our analysis.

[Kovács et al. \[2013\]](#) also present a non-interference analysis for CFGs based on abstract interpretation. Instead of concretizing to a hyper-semantics, they construct a product program [[Barthe et al. 2011](#)] that executes two instances of the original program. To this end, they define a set of rewrite rules that rely on the CFG to consist only of structured control flow (if-then-else, while). Each binary branch is expanded to four successors, one for each combination of the branch predication valuations ((true, true), (true, false), ...) which can super-linearly explode the size of the CFG. In contrast, we support reducible CFGs and our modifications (see Section 3.1) are purely technical and local to loop headers and exits which leads only to a linear worst-case increase of the CFG.

One particular property of their analysis is that for regions where two different control flow parts are paired ((true, false), (false, true)), they apply a cost metric to pair up the individual instructions of a basic block. The goal of the cost metric is to align the two programs in such a way that the abstract transformers, which operate on pairs of program points, can be made more precise by allowing information from both programs to interact. In our analysis, we cannot do something like this because we do not construct pair programs. Even more, in SPMD programs, the application scenario we focus on in this work, this is also no option, because the "pairing" is done by the hardware (GPU lanes, CPU vector registers) and cannot be influenced by the compiler.

Another line of work by [Wasserrab et al. \[2009\]](#) and [Snelting et al. \[2006\]](#) on non-interference for CFGs is based on program dependence graphs. Their work follows [Amtoft's](#) work on the correctness of slicing using program dependence [[Amtoft 2008](#)]. In their particular setting, control dependences are not considered imprecise (as opposed to the discussion in paragraph "control dependence" above) because they also account for observation: Being able to observe that a certain program point has been executed is also a disclosure of information which implies that splits that disjointly reach the end node of the CFG have to be considered. Note that this makes more splits relevant (and is therefore less precise) in our setting. Our analysis does not consider observation in this sense, can however be extended to support it in the following way: Insert a dummy variable and assignments to this variable at every program point whose execution should be observed. All splits that are relevant at the end node of the CFG for this variables coincide to the set of splits identified by these analyses. Their work comes with a mechanized [[Wasserrab et al. 2009](#)] correctness proof [[Snelting et al. 2006](#)] that relates non-interference to PDG-based backward slices. One particular difference to our approach is that, by using slicing, they implicitly rely on given data and control dependences which essentially relate to variable names (or at least storage locations). In contrast, by using abstract interpretation, our analysis relates directly to the semantics of the program. The difference is that we could combine our analysis with other analyses using standard abstract interpretation techniques (reduced products) to provide additional invariants that help to prove variables uniform.

Computing Disjoint Paths. Algorithm 1 solves the *decision* version of the k -sources two-sinks variant of the node-disjoint paths problem for DAGs, i.e. do there exist two disjoint paths from a set of k source nodes to two sink nodes? Tholey [2012] presents an algorithm that solves the 2-sources variant of this problem and shows that $O(|V| + |E|)$ is optimal. Thus, Algorithm 1 is optimal as well.

The difference between our and Tholey’s algorithm is that his algorithm needs to pre-compute a dominance tree and a shortest path tree while ours does not. However, his algorithm computes information that can answer more than the decision problem of the 2-sources variant. He can also tell which source is connected to which sink in each of the disjoint paths. Our algorithm cannot do that but it is also not relevant for our analysis.

The seminal work of Cytron et al. [1989] on SSA construction establishes a connection between dominance frontiers and join points of disjoint paths. However, in this setting, the set of join points implicitly include joins from paths that emerge from the CFG start node. Similarly, in post-dominance this criterion implicitly includes splits that lead to the end node of the CFG. Hence, techniques based on (pos-) dominance frontiers only solve a restricted case of the disjoint paths problem that leads to imprecision in our setting (see above).

9 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a formal account of an abstract interpretation for uniformity on reducible control flow graphs. Our characterization of the *relevant* control flow splits, that influence the uniformity of variables at other program points, derives from semantic considerations that emerge in the correctness proof of the abstract transformer. Our analysis is more precise than existing analyses and less constraint with respect to the control flow structure it accepts. Our formal treatment is to the most part mechanized in Coq.

Our experimental evaluation shows that the compile time and the precision of our analysis is on par with LLVM’s default divergence analysis that is only sound on syntactically more restricted CFGs. At the same time, our analysis is faster (speedup of 1.25) and achieves better precision (5% more instructions classified uniform) than a state-of-the-art non-interference analysis that is sound and least as general as our analysis.

Interesting future directions could lift the reducibility requirement on the CFG and explore the connection and applicability of our analysis to non-interference, binding time, and dependency abstract interpretations in general. Additionally, a more formal treatment of an SSA version of our analysis would be interesting because it could be connected with a more formal study of loop-closed and gated SSA forms.

ACKNOWLEDGMENTS

We thank Raimund Seidel for several insightful discussions and for giving us the hint to use Menger’s theorem to prove Lemma 5.1. We also express our gratitude to the anonymous reviewers for their detailed comments.

This work has been supported by the German federal ministry of research and by NEC.

REFERENCES

- Martin Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. 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*. 147–160. <https://doi.org/10.1145/292540.292555>
- Alexander Aiken and David Gay. 1998. 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*. 342–354. <https://doi.org/10.1145/268946.268974>

- Rajeev Alur, Joseph Devietti, Omar S. Navarro Leija, and Nimit Singhania. 2017. 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*. 507–525. https://doi.org/10.1007/978-3-319-63387-9_25
- Torben Amtoft. 2008. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.* 106, 2 (2008), 45–51. <https://doi.org/10.1016/j.ipl.2007.10.002>
- Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. 2017. Hypercollecting semantics and its application to static analysis of information flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 874–887. <https://doi.org/10.1145/3009837>
- Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. 1996. Fast, Effective Dynamic Compilation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania, USA, May 21-24, 1996*. 149–159. <https://doi.org/10.1145/231379.231409>
- Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. 1990. 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*. 257–271. <https://doi.org/10.1145/93542.93578>
- Gilles Barthe, Mathilde Duclos, and Yassine Lakhnech. 2011. A Computational Indistinguishability Logic for the Bounded Storage Model. In *Foundations and Practice of Security - 4th Canada-France MITACS Workshop, FPS 2011, Paris, France, May 12-13, 2011, Revised Selected Papers*. 102–117. https://doi.org/10.1007/978-3-642-27901-0_9
- David Bucciarelli et al. 2020. LuxMark, v3.1. https://wiki.luxcorerenderer.org/LuxMark_v3. Accessed: 2020.06.30.
- Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, Thomas Raoux, and Konrad Trifunovic. 2019. 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*. 254–265. <https://doi.org/10.1109/CGO.2019.8661189>
- Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210. <https://doi.org/10.3233/JCS-2009-0393>
- Sylvain Collange. 2011. *Identifying scalar behavior in CUDA kernels*. Technical Report. ENS Lyon. <https://hal.archives-ouvertes.fr/hal-00555134/>
- Charles Consel. 1990. Binding Time Analysis for High Order Untyped Functional Languages. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*. ACM, 264–272. <https://doi.org/10.1145/91556.91668>
- Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. 2001. *A Simple, Fast Dominance Algorithm*. Technical Report TR-06-33870. Rice University.
- Patrick Cousot. 2019. Abstract Semantic Dependency. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 389–410. https://doi.org/10.1007/978-3-030-32304-2_19
- Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. 238–252. <https://doi.org/10.1145/512950.512973>
- Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. <https://doi.org/10.1145/567752.567778>
- Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintão Pereira, and Wagner Meira Jr. 2011. Divergence Analysis and Optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011, Galveston, TX, USA, October 10-14, 2011*. 320–329. <https://doi.org/10.1109/PACT.2011.63>
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*. ACM Press, 25–35. <https://doi.org/10.1145/75277.75280>
- Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490. <https://doi.org/10.1145/115372.115320>
- Craig A. Farrell and Dorota H. Kieronska. 1996. Formal Specification of Parallel SIMD Execution. *Theor. Comput. Sci.* 169, 1 (1996), 39–65. [https://doi.org/10.1016/S0304-3975\(96\)00113-2](https://doi.org/10.1016/S0304-3975(96)00113-2)
- Axel Habermaier and Alexander Knapp. 2012. 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.* 316–335. https://doi.org/10.1007/978-3-642-28869-2_16
- Paul Havlak. 1993. 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.* 477–499. https://doi.org/10.1007/3-540-57659-2_28
- Matthew S. Hecht and Jeffrey D. Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (1974), 367–375. <https://doi.org/10.1145/321832.321835>
- Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A Performance-Portable OpenCL Implementation. *Int. J. Parallel Program.* 43, 5 (2015), 752–785. <https://doi.org/10.1007/s10766-014-0320-y>
- Neil D. Jones, Peter Sestoft, and Harald Søndergaard. 1989. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation* 2, 1 (1989), 9–50.
- Guido Juckeland, William C. Brantley, Sunita Chandrasekaran, Barbara M. Chapman, Shuai Che, Mathew E. Colgrove, Huiyu Feng, Alexander Grund, Robert Henschel, Wen-mei W. Hwu, Huian Li, Matthias S. Müller, Wolfgang E. Nagel, Maxim Perminov, Pavel Shelepugin, Kevin Skadron, John A. Stratton, Alexey Titov, Ke Wang, G. Matthijs van Waveren, Brian Whitney, Sandra Wienke, Rengan Xu, and Kalyan Kumaran. 2014. SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation - 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers.* 46–67. https://doi.org/10.1007/978-3-319-17248-4_3
- Ralf Karrenberg. 2015. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs.* Springer. <https://doi.org/10.1007/978-3-658-10113-8>
- Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. 2013. Relational abstract interpretation for the verification of 2-hypersafety properties. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013.* 211–222. <https://doi.org/10.1145/2508859.2516721>
- Christian Lattner. 2004. *Loop Optimizer Notes.* Retrieved November 20, 2019 from <http://nondot.org/sabre/LLVMNotes/LoopOptimizerNotes.txt>
- Chris Lattner and Vikram S. Adve. 2004. 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.* 75–88. <https://doi.org/10.1109/CGO.2004.1281665>
- Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanovic. 2013. 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.* 32:1–32:11. <https://doi.org/10.1109/CGO.2013.6494995>
- Roland Leiða, Immanuel Haffner, and Sebastian Hack. 2014. 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.* 17–24. <https://doi.org/10.1145/2568058.2568062>
- Yun Liang, Muhammad Teguh Satria, Kyle Rupnow, and Deming Chen. 2016. An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems* 35, 7 (2016), 1165–1178. <https://doi.org/10.1109/TCAD.2015.2501303>
- Erik Lindholm, John Nickolls, Stuart F. Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55. <https://doi.org/10.1109/MM.2008.31>
- Taylor Lloyd, Karim Ali, and José Nelson Amaral. 2019. GPUcheck: Detecting CUDA Thread Divergence with Static Analysis. (2019).
- Jan Christian Menz. 2016. A Coq Library for Finite Types. *Bachelor's thesis, Universität des Saarlandes* (2016).
- Simon Moll and Sebastian Hack. 2018. 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.* 543–556. <https://doi.org/10.1145/3192366.3192413>
- Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg.
- Hanne Riis Nielson and Flemming Nielson. 1988. Automatic Binding Time Analysis for a Typed lambda-Calculus. *Sci. Comput. Program.* 10, 1 (1988), 139–176. [https://doi.org/10.1016/0167-6423\(88\)90025-1](https://doi.org/10.1016/0167-6423(88)90025-1)
- NVIDIA. 2017. V100 GPU architecture. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>. Accessed: 2019.11.20.
- Arsène Pérard-Gayot, Richard Membarth, Roland Leiða, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: generating renderers without writing a generator. *ACM Trans. Graph.* 38, 4 (2019), 40:1–40:12. <https://doi.org/10.1145/3306346.3322955>
- Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012. IEEE,* 1–13.

- Sebastian Pop, Pierre Jouvelot, and Georges-André Silber. 2009. In and Out of SSA: A Denotational Specification. (2009). <http://cri.ensmp.fr/classement/doc/E-285.pdf>
- Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. 2017. Auto-vectorization for image processing DSLs. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2017, Barcelona, Spain, June 21-22, 2017*, Vijay Nagarajan and Zili Shao (Eds.). ACM, 21–30. <https://doi.org/10.1145/3078633.3081039>
- Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. 2013. Divergence analysis. *ACM Trans. Program. Lang. Syst.* 35, 4 (2013), 13:1–13:36. <https://doi.org/10.1145/2523815>
- Sigurd Schneider. 2018. *A verified compiler for a linear imperative / functional intermediate language*. Ph.D. Dissertation. Saarland University, Saarbrücken, Germany. <https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/27296>
- Alexander Schrijver. 2017. *A Course in Combinatorial Optimization*. Retrieved November 19, 2019 from <https://homepages.cwi.nl/~lex/files/agt3.pdf> III. Disjoint paths, Theorem 3.
- Gregor Snelting, Torsten Robschink, and Jens Krinke. 2006. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.* 15, 4 (2006), 410–457. <https://doi.org/10.1145/1178625.1178628>
- Yifan Sun, Xiang Gong, Amir Kavayan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David R. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. 13–22. <https://doi.org/10.1109/IISWC.2016.7581262>
- Torsten Tholey. 2012. Linear time algorithms for two disjoint paths problems on directed acyclic graphs. *Theor. Comput. Sci.* 465 (2012), 35–48. <https://doi.org/10.1016/j.tcs.2012.09.025>
- John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. 2014a. Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations. In *Solving Software Challenges for Exascale - International Conference on Exascale Applications and Software, EASC 2014, Stockholm, Sweden, April 2-3, 2014, Revised Selected Papers*. 39–56. https://doi.org/10.1007/978-3-319-15976-8_3
- John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014b. XSbench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. Kyoto. <https://www.mcs.anl.gov/papers/P5064-0114.pdf>
- Peng Tu and David A. Padua. 1995. 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*. 47–55. <https://doi.org/10.1145/207110.207115>
- Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 683–710. https://doi.org/10.1007/978-3-319-89884-1_24
- Daniel Wasserrab, Denis Lohner, and Gregor Snelting. 2009. 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*. 31–44. <https://doi.org/10.1145/1554339.1554345>