Branch Divergence in SIMT Machines:

From Control-Flow Graph Theory to Compiler Divergence Analysis in LLVM

Mahesha S
Compiler Engineer

mahesha.shivamallappa@gmail.com

Abstract

General-purpose GPU (GPGPU) programs are written in a Single-Program, Multiple-Data (SPMD) style, yet are executed by hardware in a Single-Instruction, Multiple-Thread (SIMT) fashion: groups of scalar threads, called warps or wavefronts, are marched in lock-step through the same instruction stream on a shared program counter. This mismatch between the programming model and the execution model gives rise to *branch divergence* – the phenomenon in which threads of the same warp disagree on the outcome of a conditional branch and must be serialized. This paper develops the branch-divergence problem from first principles. We begin with the classical control-flow-graph theory of Allen, Hecht, Ullman, and Tarjan – reducibility, interval analysis, and structured versus unstructured control flow – and show why these forty-five-year-old ideas are exactly the right lens for understanding modern SIMT hardware. We then describe how SIMT hardware reconverges divergent threads at the immediate post-dominator of a divergent branch using a reconvergence stack, why unstructured and irreducible control flow defeats this mechanism, and the two lines of compiler defense that have been built against it: control-flow linearization/structurization, and *divergence analysis* – the static classification of program values and branches as *uniform* or *divergent*. We survey the divergence-analysis literature (Coutinho et al., Sampaio et al., Karrenberg and Hack, Rosemann, Moll and Hack, and Sahasrabuddhe et al.) and then give a detailed, patch-level account of how this line of research was absorbed into LLVM across three successive implementations – from the original 2015 graph-reachability analysis, through Simon Moll's 2018 generic sync-dependence framework restricted to reducible graphs, to the current (2022) cycle-based Uniformity Analysis that soundly handles irreducible control flow on both LLVM IR and Machine IR.

Keywords: SIMT execution, branch divergence, control-flow graph reducibility, control-flow linearization, divergence analysis, uniformity analysis, LLVM, GPU compilers.

Contents

1 Introduction

Graphics Processing Units (GPUs) have become the default engine for data-parallel computation, from rendering triangles to training transformers. What makes them fast is also what makes them fragile: a GPU executes a huge number of scalar threads under an execution discipline called **SIMT – Single Instruction, Multiple Thread** – in which threads are grouped into small squads (called **warps** or **wavefronts**, depending on the hardware) and marched through the *same* instruction, on the *same* cycle, using a *single, shared* program counter, with each individual thread of such a squad called a **lane**. It is, in effect, SIMD execution wearing a thread's clothing. This paper sticks with the vendor-neutral term **SIMT group** for these squads.

Programmers, however, do not write SIMT code. They write SPMD code – Single Program, Multiple Data – in which every thread is described as an independent sequential program operating on its own slice of data, free to take its own branches, its own loop trip counts, its own path through the code. The GPU compiler and the GPU hardware jointly carry the burden of reconciling these two views: the programmer's illusion of thousands of independent threads, and the hardware's reality of lock-step execution of these threads as SIMT groups.

The seam between these two models often shows up wherever a *conditional branch* inside SIMT group can be taken differently by different lanes. This is **branch divergence**. When it happens, the hardware cannot let two lanes of the same SIMT group – suppose, for concreteness, that lane number 3 wants the `then` side while lane number 5 wants the `else` side – run down different sides of the branch in the same cycle: there is only one program counter to share. Instead, the hardware serializes: it runs the `then` path with the lanes that wanted it active and the `else`-lanes masked off, then runs the `else` path with the roles reversed, and finally **reconverges** all lanes once they reach a common point. Whatever parallelism existed across the two paths is lost. More generally, a single branch (e.g. a `switch` statement) can send the lanes of one SIMT group down n distinct targets, where n is at most the size of the SIMT group (32 or 64, on current hardware); such an n-way divergent branch is executed as n separate serialized passes and can therefore cost up to $n\times$ the work of a non-divergent one. This serialization is a direct source of *performance degradation*: cycles that could have advanced every lane of the SIMT group together are instead spent advancing only the subset of lanes active on one path at a time, while the rest sit idle behind their mask.

Branch divergence is not merely a performance curiosity, however – in early, naive hardware/-compiler co-designs, badly structured control flow could even produce *incorrect* results, because the assumption that all active lanes reconverge at a well-defined program point silently breaks down for control-flow graphs that lack that structure. This is the thread this paper pulls on. We ask, and answer, four questions in sequence:

1. **What property of a program's control-flow graph determines whether hardware reconvergence works at all?** The answer, it turns out, was worked out for entirely different reasons – sequential compiler optimization – in the 1970s, by Frances Allen, Matthew Hecht, Jeffrey Ullman, and Robert Tarjan, under the names *reducibility* and *structured control flow*. Section 2 develops this theory from first principles because it is the load-bearing foundation for everything that follows.

2. **Given that theory, how exactly does SIMT hardware reconverge divergent threads, and what happens when the control-flow graph does not cooperate?** Section 3 and Section 4 answer this with a worked example of the classical reconvergence-stack mechanism, and survey the two hardware responses in the literature – stack-based reconvergence at the immediate post-dominator, and dynamic warp formation.

3. **What can the *compiler* do about it?** Section 5 covers the first line of compiler defense: rewriting the control-flow graph itself, via *structurization* and *linearization*, so that hardware reconvergence has a chance to work well. Section 6 covers the second, more surgical line of defense: *divergence analysis*, a static analysis that classifies every value and branch in the program as **uniform** (same value/outcome on every active lane) or **divergent**, so that the compiler applies expensive divergence-mitigating transformations only where they are actually needed, and can perform new optimizations (scalarization, uniform branch simplification) where they are not.

4. **How did this research actually make it into a production compiler?** Section 7 is a patch-level history of divergence/uniformity analysis inside LLVM, from the original 2015 implementation, through Simon Moll's 2018 redesign, to the 2022 cycle-based Uniformity Analysis that is the current default – three successive implementations, each motivated by a concrete limitation of its predecessor, each traceable to a specific, citable source: an academic paper for the first, and public LLVM RFCs for the second and third.

Takeaway: What this paper is

This is a tutorial survey, organized as a single, continuous argument running from 1970s dataflow-analysis theory to a 2022 LLVM patch. No single piece of the technical content is new; the contribution is in connecting control-flow-graph reducibility, SIMT hardware reconvergence, and compiler divergence analysis into one coherent narrative, with a level of implementation detail (LLVM class names, differential-revision numbers, dates) that is not usually collected in one place.

Notation

Throughout, we write a control-flow graph as a pair (G, s), where G is a directed graph over basic blocks and s is its unique start (entry) node; every node is assumed reachable from s. We write $\text{IPDOM}(b)$ for the immediate post-dominator of block b: the closest block that lies on every path from b to the (unique) exit. We use **lane** for a single SIMT thread's slot within a SIMT group, and **active mask** for the bit-vector recording which lanes currently participate in execution.

2 Control-Flow Graphs: The Load-Bearing Theory

Everything in this paper – hardware reconvergence, control-flow linearization, divergence analysis – is a statement about the *shape* of a program's control-flow graph. Before we can say anything about SIMT hardware, we need a small, precise vocabulary for that shape. This section develops it, following the classical treatments of Allen [2], Hecht and Ullman [13, 14], and Tarjan [34].

2.1 Control-Flow Graphs, Dominance, and Post-Dominance

A **control-flow graph** (CFG) of a procedure P is a directed graph (G, s) whose nodes are the basic blocks of P and whose edges represent possible transfers of control; s is the unique entry block, and we assume (by convention) a unique exit block e that every path eventually reaches. Two relations on this graph are central to everything that follows in this paper:

Definition: Dominance

Block d **dominates** block b (written $d \succeq b$) if every path from s to b passes through d. Block d is the **immediate dominator** of b, written $\text{idom}(b)$, if $d \succeq b$, $d \neq b$, and d does not dominate any other dominator of b other than itself – i.e. $\text{idom}(b)$ is the closest strict dominator of b.

Definition: Post-Dominance

Block p **post-dominates** block b (written $p \preceq b$) if every path from b to e passes through p. The **immediate post-dominator** of b, written $\text{IPDOM}(b)$, is the closest strict post-dominator of b: the first block that *every* continuation of b is guaranteed to reach.

Post-dominance is the more important of the two relations for this paper. Intuitively, $\text{IPDOM}(b)$ answers the question – "no matter which way control leaves b, where is it guaranteed to end up eventually?" – and that question is exactly the one SIMT hardware needs answered every time a branch diverges (Section 4). Both relations can be computed in $O(n \log n)$ (and, with more machinery, near-linear) time by the classical Lengauer–Tarjan algorithm [20]; a Lengauer–Tarjan pass on the *reverse* CFG computes post-dominance.

2.2 Structured versus Unstructured Control Flow

Definition: SESE Region

A **Single-Entry, Single-Exit** (SESE) region of a CFG is a connected subgraph with exactly one edge entering it from the outside and exactly one edge leaving it to the outside.

Definition: Structured Control Flow

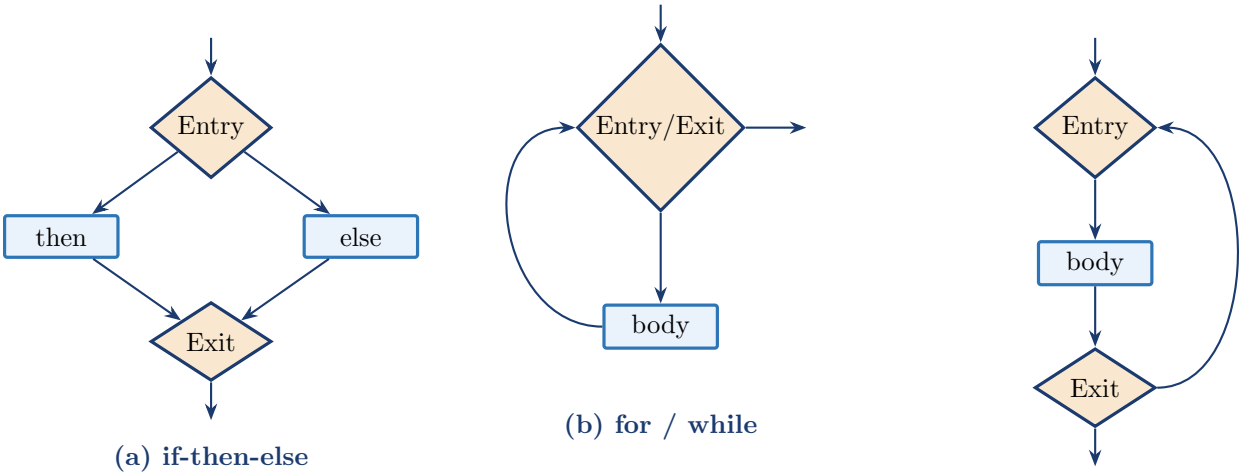
A CFG (G, s) exhibits **structured control flow** if every region of G – every `if`, `if-else`, `while`, `do-while`, and their nestings – is a SESE region. If some region violates this (some construct has more than one way in, or more than one way out), G is **unstructured**.

Figure 1 shows the canonical structured shapes: an `if-then-else`, a `for`/`while` loop, and a `do-while` loop. Each is SESE by construction, and each is what a recursive-descent parser for a structured language (C, C++, OpenCL, CUDA C++) naturally emits, provided the programmer does not reach for unstructured escape hatches.

Figure 2 gives the C source behind each shape in Figure 1, so that the graph and the code that produces it can be read side by side.

Figure 3 shows how easily this property is lost. A single stray edge is enough: an early `goto` that gives the `then` arm a second exit, a `goto` issued before the branch is even reached that gives the `else` arm a second entry, or a `break` that gives a loop body a second exit straight to the loop's exit port. In C-family languages, the usual sources of unstructured control flow are [35]: (i) undisciplined use of `goto`; (ii) short-circuited boolean expressions (`&&`, `||`) lowered to branches by the front-end; and (iii) compiler transformations themselves – jump threading, tail duplication, loop rotation, and switch-to-branch lowering all routinely introduce extra edges that break SESE-ness, even when the source program was perfectly structured.

Figure 4 shows the C source behind each of the three breakages in Figure 3: in every case, a single stray `goto` (or `break`, which is just a disguised `goto` to the loop's exit point) is enough to turn a SESE region into a non-SESE one.



(a) if-then-else (b) for / while (c) do-while

Figure 1: The three canonical structured shapes, each drawn with its single external entry edge (arriving from above) and single external exit edge (leaving below, or to the right for the Entry/Exit node in (b)) made explicit. Every region here has exactly one entry edge and one exit edge – the defining SESE property.

```
if (cond) {
    ...
} else {
    ...
}
```

(a) if-then-else

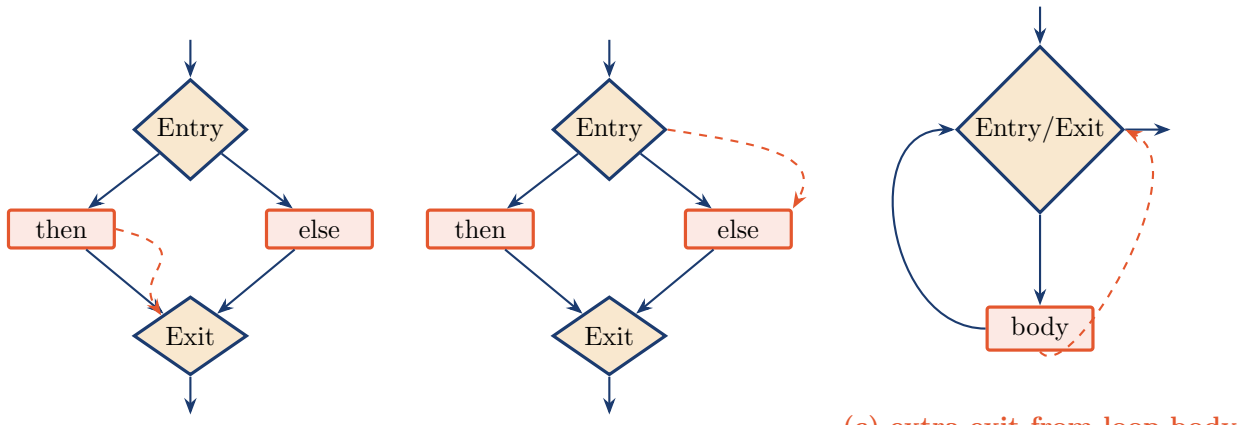
```
while (cond) {
    ...
}
```

(b) for / while

```
do {
    ...
} while (cond);
```

(c) do-while

Figure 2: C source producing each of the three structured shapes in Figure 1. Every basic block of the resulting CFG still has exactly one way in and one way out, because the source has no `goto`.



(a) extra exit from `then` (b) extra entry into `else` (c) extra exit from loop body

Figure 3: Three ways structured control flow breaks: (a) a stray outgoing edge from inside an if-then-else, (b) a stray incoming edge into one of its arms, and (c) a stray outgoing edge from a loop body. Each dashed (coral) edge is enough to destroy the SESE property of the enclosing region.

```c
if (cond) {
    ...
    /* extra */
    goto L;
} else {
    ...
}
L: ...
```

(a) extra exit from `then`

```c
if (special)
    /* extra */
    goto L;
...
if (cond) {
    ...
} else {
L:  ...
}
```

(b) extra entry into `else`

```c
while (cond) {
    ...
    if (x)
        /* extra */
        break;
    ...
}
```

(c) extra exit from loop body

Figure 4: C source producing each of the three breakages in Figure 3. In (a), `then`'s early `goto` gives it a second exit; in (b), the entry block itself contains a second, unconditional route into `else` – via a `goto` issued before the `if` is even reached – so `else` ends up entered both from that `goto` and from the ordinary false branch; in (c), the `break` gives the loop body a second exit – straight to the loop's exit port – alongside its normal fall-through back to the header.

2.3 Reducibility

Structuredness, as just defined, is a syntactic, region-based property: it asks whether the source-level nesting of `if`/`while` blocks stays SESE all the way down. **Reducibility** asks a related but different question, this time purely about the graph itself, with no reference to source syntax at all: *does every loop have a single, well-defined entry point?* We build up the answer in three steps: first a dominance-based picture of what a "well-behaved" loop is (Section 2.3.1); then the graph-theoretic machinery that turns that picture into a precise, checkable property, from two independent angles (Section 2.3.2); and finally how the two views – structuredness and reducibility – actually relate to each other, which turns out to be a one-directional implication rather than an equivalence (Section 2.3.3).

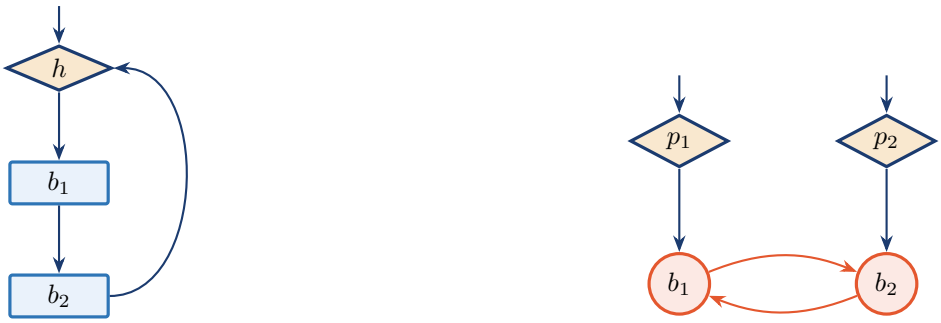
2.3.1 Natural Loops: A Dominance-Based View

Recall from Section 2.1 that block d dominates block b if every path from the start reaches b through d. A **natural loop** is exactly what dominance is built to describe: a set of blocks that all loop back to a single **header** h, where h dominates every other block in the set – so that h is the *only* block anything outside the loop can jump into (Figure 5(a)). Every loop a structured-language compiler emits (Figures 1 and 2) is a natural loop in exactly this sense: the `while`/`for` test and the `do-while` header are, by construction, the sole point of entry.

The failure mode is just as easy to state in the same vocabulary: a loop-like, mutually-reachable region stops being a natural loop the moment it can be entered from two or more *incomparable* blocks – blocks where neither dominates the other – because then no single block is left that can be called *the* header (Figure 5(b)). This is precisely what a stray `goto` that lands inside a loop from outside it (rather than at the loop's own header) or a later compiler pass merging two separately-structured loops into one can produce.

2.3.2 Reducible and Irreducible Flow Graphs

The literature gives two independent, equivalent ways to turn the natural-loop intuition above into a precise, checkable graph property.



(a) natural loop **(b) unnatural loop**

Figure 5: (a) A natural loop: header h has the loop's only external entry, and dominates every other block in it, so b_1, b_2 are only reachable by first passing through h. (b) An unnatural loop: two incomparable predecessors p_1, p_2 enter the mutually-reachable pair b_1, b_2 at two different points (with separate edges $b_1 \to b_2$ and $b_2 \to b_1$ forming the mutual loop), so no single block dominates the whole region – there is no header left to call *the* entry.

Angle 1: interval analysis.

Allen's original formulation [2] is algorithmic. An *interval* with header h is the maximal single-entry subgraph containing h such that every closed path (cycle) through any node of the subgraph also passes through h. Interval analysis repeatedly (1) partitions the current graph into intervals and (2) collapses each interval to a single node, producing a smaller **derived graph**, and repeats on the derived graph.

Definition: Reducible / Irreducible Flow Graph

Given (G, s): if repeated interval analysis reduces G all the way down to a single node, G is a **reducible** flow graph. If interval analysis makes partial progress but gets stuck above a single node, G is **non-reducible**. If interval analysis cannot even make partial progress (the very first derived graph equals G), G is **irreducible** in the strict sense. In practice – and throughout the rest of this paper – we follow common usage and collapse **non-reducible** and **irreducible** into a single umbrella term, **irreducible**, since the practical consequences for compilers and SIMT hardware are the same either way.

This interval-based definition is provably equivalent to the natural-loop picture of Section 2.3.1: G is reducible if and only if every loop in G is a natural loop, i.e. has a single entry point that dominates every other node in the loop. Compiler engineers reach for this operational form far more often than for interval analysis itself, since it can be checked directly against a dominator tree once one has already been computed for other purposes.

Angle 2: the Hecht–Ullman structural characterization.

The second angle sharpens the same property into an exact structural pattern, rather than the outcome of an iterative procedure:

Theorem: $(*)$-Characterization

A flow graph is **non-reducible** if and only if it contains, as a subgraph, the pattern shown in Figure 6 (traditionally called the **$(*)$-subgraph**): a header n_0 reaching a node a, from which

two disjoint paths reach two nodes b and c that *mutually* reach each other, so that neither b nor c dominates the other, yet both are reachable by more than one route from a. This result is due to Hecht and Ullman [14].

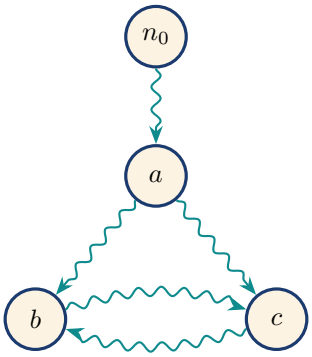


Figure 6: The $(*)$-subgraph: the minimal irreducibility witness of Hecht and Ullman [14]. Node a can reach both b and c, and b, c mutually reach each other without either dominating the other – there is no single header that dominates this whole loop-like region.

Testing whether a given flow graph is reducible can be done in near-linear time [34]; Tarjan's algorithm and its refinements (surveyed by Offner [25]) are what compilers use in practice rather than literally running interval analysis to completion or searching for $(*)$-subgraphs by brute force.

2.3.3 Connecting Structuredness and Reducibility

With both views in hand, we can now state exactly how Section 2.2's syntactic property (structured control flow) and this section's graph-theoretic property (reducibility) relate: as a one-directional implication.

Theorem: Structuredness Implies Reducibility

If a CFG (G, s) exhibits structured control flow, then G is a reducible flow graph.

Why it holds. Structuredness forces every loop to be a SESE region entered only through its own header. That is exactly the definition of a natural loop (Section 2.3.1), and a graph in which every loop is a natural loop is exactly what reducibility requires. So structured control flow is sufficient for reducibility.

Why the converse does not hold. The converse statement would be: *if G is a reducible flow graph, then G exhibits structured control flow.* This is false. The reason is that reducibility is a purely *loop-centric* property: it only constrains how loops are entered. Structuredness is stricter – it also constrains how `if`-regions are entered and exited – so a CFG can be unstructured without ever touching a loop header, and remain fully reducible. Figures 3 and 4 are three such graphs: the extra exit from `then`, the extra entry into `else`, and the extra loop exit via `break` are all unstructured, yet none of them creates a second entry point into any loop – the first two involve no loop at all, and the third only adds an extra *exit* from the loop body, leaving its single header untouched. All three are therefore unstructured but still reducible. The only thing that breaks reducibility is a `goto` of a more specific kind: one that jumps into a loop from *outside* it, landing somewhere other than the loop's header, producing the $(*)$-subgraph of Figure 6. Structuredness rules this out along

with everything else; reducibility only rules out this one case. Hence structured ⇒ reducible, but not the other way around.

Observation: Irreducibility Is Rare, But Consequential

In well-written, structured source programs, irreducible control flow is uncommon. But it is not *absent*: it is introduced by (i) genuine multi-entry loops written with `goto` (rare in practice but legal C), and, far more commonly in a compiler pipeline, (ii) control-flow merging transformations, most importantly for our purposes the *control-flow structurization/linearization* passes discussed in Section 5, which can convert one shape of irregularity into another, and (iii) irreducibility introduced by *other* optimization passes (jump threading, loop rotation, cross-jumping) upstream of the code generator, exactly as noted for source-level unstructuredness above. Reducibility matters more than structuredness for the rest of this paper because reducible loops admit efficient, precise dataflow analysis (interval-based iterative algorithms converge quickly and can be shown to terminate in a bounded number of passes over the loop nesting structure), and, as we will see in Sections 4 and 7, they are exactly the condition under which the classical SIMT reconvergence mechanism – and its compiler-side analyses – were originally designed to work.

2.4 Regions, Program Structure Trees, and Related Graph-Theoretic Tools

Two further classical tools recur throughout the divergence literature. The **Program Structure Tree** of Johnson, Pearson, and Pingali [16] computes all SESE regions of a CFG in linear time and organizes them into a tree ordered by nesting – an efficient, purely graph-theoretic replacement for a parser's lexically-nested view of `if`/`while` blocks, and a natural substrate for control-flow restructuring algorithms (Section 5). **Hammock graphs** [37] are a related, slightly coarser structuring device used by several GPU compilers to identify SESE "hammock" regions suitable for predication. Finally, Allen, Kennedy, Porterfield, and Warren's classical **control-dependence to data-dependence conversion** technique [3] – turning a branch's influence on *which statements execute* into an explicit data value that downstream analyses can propagate like any other – is the direct conceptual ancestor of the **sync-dependence** construct used by every divergence analysis in Section 6.

With this vocabulary in hand – CFGs, dominance and post-dominance, structured versus unstructured regions, and reducible versus irreducible graphs – we can now say precisely what SIMT hardware assumes about a program's control flow, and precisely how that assumption can fail.

3 From SPMD Programs to SIMT Execution

3.1 The Programmer's View: SPMD

A GPGPU kernel – written in CUDA C++, OpenCL C, or a similar data-parallel dialect – is launched as a very large number of scalar **threads**, each running the exact same kernel function body, each indexed by a built-in thread identifier (`threadIdx` in CUDA, `get_global_id()` in OpenCL).

From the programmer's point of view:

- every thread is logically independent of every other thread;

- every thread executes its own, complete instance of the kernel, on its own slice of data, and is free to take its own path through `if` statements and loops based on its own thread index and its own data;
- there is no hardware-visible notion of "groups" of threads that must agree with one another.

This is the classical **Single Program, Multiple Data** (SPMD) programming model. It is a convenient fiction, and the entire point of this paper is what happens when hardware makes that fiction efficient to execute.

3.2 The Hardware's View: SIMT

At the hardware level, threads are not scheduled individually. They are grouped, statically, into fixed-size squads – typically 32 or 64 threads, called a **warp** or a **wavefront** depending on the hardware – and every thread inside one squad executes in SIMD lock-step: the same instruction, fetched and decoded once, fed to all of the squad's arithmetic lanes on the same cycle, from a single shared program counter. This execution discipline is **SIMT**: *Single Instruction, Multiple Thread*. Every lane that is not supposed to participate in a given instruction (because it took a different branch, or has already exited) is simply masked off – it receives the instruction but its result is discarded and its architectural state left unchanged.

Takeaway: The Central Tension

GPU hardware maps the programmer's SPMD model onto a SIMT execution substrate *at run time*, by grouping threads into SIMT groups that must share one program counter. Everything this paper examines – from the hardware's reconvergence machinery to the compiler's control-flow transformations and divergence analysis – exists to manage the gap between "every thread is independent" (what the programmer assumes) and "every thread in a SIMT group shares a PC" (what the hardware provides).

3.3 Why SIMT? The Hardware's Rationale

The previous two subsections described *what* the programmer writes and *what* the hardware does with it; this one asks *why* the hardware does it that way. SPMD is a free, natural abstraction for the programmer – describe one thread's work, launch a huge number of copies – but it says nothing about how those threads should actually be executed in silicon. The most literal implementation would give every thread its own fetch, decode, and branch-prediction logic and its own program counter: true MIMD execution, exactly like an ordinary multicore CPU scaled up to thousands of cores. GPUs do not do this, and the paper that introduced the term SIMT in the first place [21] is explicit that the reason is not philosophical, it is economic: die area and power.

On a modern CPU core, the control logic that decides *what to run next* – instruction fetch, decode, branch prediction, out-of-order scheduling – costs far more silicon and power than the arithmetic units that do the actual work. Replicating that control logic once per thread, tens of thousands of times over, to give every SPMD thread its own independent program counter is simply not affordable at GPU thread counts. SIMT is the alternative: amortize the control logic across many threads instead of replicating it per thread. One fetch, decode, and branch unit is shared across an entire SIMT group of 32 or 64 lanes; only the cheap part – an ALU lane and its private register file – is replicated per thread. A control-logic cost that would otherwise grow with the number of threads instead stays fixed per group, and the area and power this frees up buys two

things: more arithmetic lanes per chip, and more SIMT groups that can be resident on the chip at once.

That second payoff is what makes SIMT a genuinely different design philosophy from a CPU, not just a cheaper one – the **latency-oriented** versus **throughput-oriented** distinction that Kirk and Hwu use to organize an entire textbook [19], and that Garland and Kirk lay out directly [10]. A CPU hides memory and instruction latency with large caches, deep speculative pipelines, and branch prediction – silicon spent *avoiding* stalls. A GPU instead hides latency by keeping so many independent SIMT groups resident that when one stalls on a memory access, the scheduler simply switches, in a single cycle and at zero cost, to another group that is ready to run. This only pays off if the chip can afford to keep many groups resident at once, which is only affordable because grouping threads under SIMT has already amortized away the per-thread control overhead that would otherwise have consumed that same area and power budget.

Takeaway: Why Hardware Chose SIMT

SIMT is not the natural execution model for SPMD threads; it is the cheapest one a chip can actually build at GPU thread counts. Sharing one fetch/decode/branch unit across a whole SIMT group turns per-thread control logic from the dominant hardware cost into a rounding error, and the silicon and power this frees up buys both more arithmetic lanes and more concurrently resident SIMT groups – the latter being how GPUs hide memory latency in the first place, by switching groups instead of waiting. Branch divergence, the subject of the rest of this paper, is the price of admission for this efficiency: the very sharing that makes SIMT so cheap when threads agree is exactly what breaks down when they don't.

3.4 A Worked Example

Consider the following minimal kernel fragment, already lowered to LLVM IR, which branches on whether the calling thread's ID is zero:

Listing 1: A trivial divergent kernel fragment in LLVM IR.

```
%cond = icmp eq i32 %tid, 0
br i1 %cond, label %then, label %else
then:
  br label %merge
else:
  br label %merge
merge:
  %a = phi i32 [ 0, %then ], [ 1, %else ]
```

A large group of threads executes this program. From the programmer's SPMD vantage point, each thread independently decides, based on its own `%tid`, whether to go through `then` or `else`, and each computes its own private `%a`. At the hardware level, a SIMT group containing many threads with different `%tid` values reaches the `br i1 %cond` instruction with a mix of `true` and `false` values for `%cond` across its lanes – and this is precisely a **divergent branch**: the SIMT group's single program counter cannot simultaneously be at both `then` and `else`.

Figure 7 depicts the general shape of a SIMT group: a set of lanes sharing one instruction stream (one common PC), each lane carrying its own private register state.

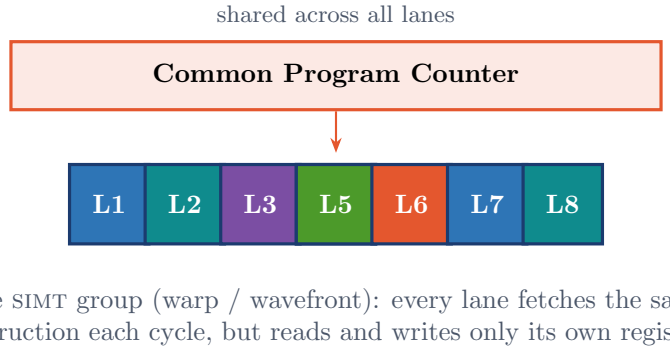


Figure 7: A SIMT group: many lanes, one shared program counter. Eight lanes are shown purely for illustration – a real SIMT group is 32 or 64 lanes wide (Section 3.2), the picture just does not scale that far. Colors distinguish lanes only for visual clarity – architecturally all lanes are identical.

4 Branch Divergence and Its Hardware Handling

4.1 Defining Branch Divergence

Definition: Branch Divergence

Consider a SIMT group executing a conditional branch whose condition is computed independently, per lane. If at least two active lanes of the group disagree on the branch outcome, the branch is said to **diverge**, and the group itself is said to be in a **divergent** state until all its lanes reconverge.

Figure 8 shows the simplest possible instance: a SIMT group of 8 lanes flows uniformly through block P into a branch at Q; some lanes take path A (toward R), the rest take path B (toward S); both paths rejoin at T. Every lane executes P and T, and each lane individually executes exactly one of R or S, never both. Yet because R and S sit on the shared instruction stream of one SIMT group, the *group* as a whole must, at the hardware level, visit both.

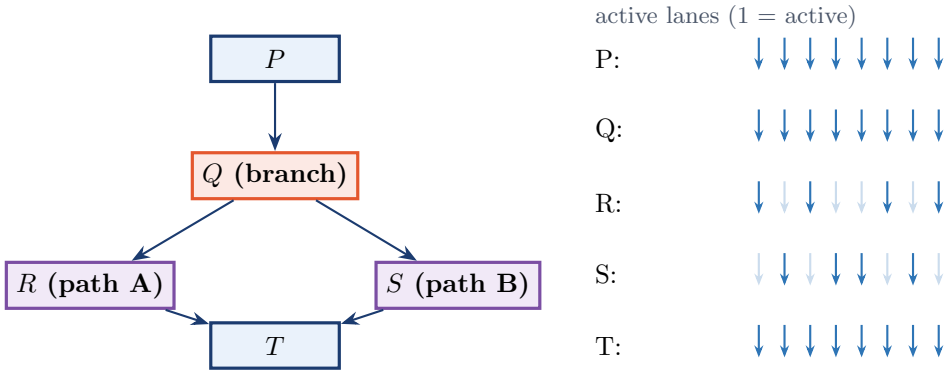


Figure 8: A minimal divergent branch. Eight lanes are drawn purely for illustration – as in Figure 7, a real SIMT group is 32 or 64 lanes wide (Section 3.2). All lanes are active through P and Q; at Q the group splits by active mask (light arrows = inactive lanes) into path A (through R) and path B (through S); all lanes reconverge at T.

Because Q's single program counter cannot simultaneously fetch instructions from both R and

S, the hardware **serializes** the two paths: it runs R to completion with only path A's lanes active and path B's lanes masked off (idle, contributing no work), then runs S with the roles reversed, and only afterward resumes with the full, reconverged 8-lane group at T. Figure 9 makes this execution order explicit as a timeline: P, Q, and T run at full, 8-lane occupancy, while R and S each run at half occupancy, one after the other.

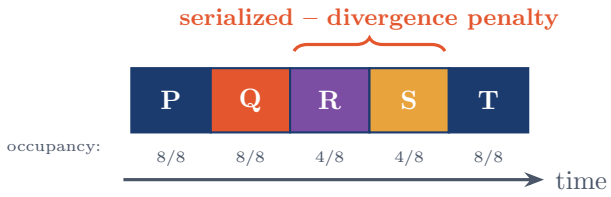


Figure 9: Execution timeline for Figure 8. P, Q, and T run at full, 8-lane occupancy; R and S each run at half occupancy (4 of 8 lanes), back to back, because the group's single program counter cannot be at both simultaneously – this back-to-back execution is exactly what **serialization** means at the hardware level.

This raises the obvious question the rest of this section answers: how does the hardware actually know when it is safe to stop running R and S back to back, and resume the full group together again at T?

4.2 SIMT Reconvergence at the Immediate Post-Dominator

The question every SIMT core must answer, the instant a branch diverges, is: *when is it safe to stop serializing and resume running all lanes together?* The classical answer, first crystallized in detail by Fung et al. [8, 9] and, for most of the GPU era, the standard reconvergence discipline across SIMT hardware generally, is:

Observation: The Reconvergence Point

The immediate post-dominator of a diverged branch Q, IPDOM(Q), is the first program point that *every* continuation of Q is guaranteed to reach (Section 2.1). It is therefore the earliest point at which the hardware can, with a purely static, one-time analysis at compile time, guarantee that reconverging all lanes is both safe (no lane is forced through code it should not execute) and complete (no lane is left behind). In Figure 8, IPDOM(Q) = T.

Between Q and IPDOM(Q), the hardware sequentially executes each diverged path with a private **active mask**: it runs path A with only A's lanes enabled and B's lanes disabled (their results discarded, their register state preserved), then runs path B with the roles reversed, and only once both paths have been drained does it resume with the full, reconverged mask at IPDOM(Q).

The classical implementation mechanism is a small hardware stack of *(reconvergence PC, next PC, active mask)* triples, per SIMT group, maintained by the branch/reconvergence unit. We illustrate it with the nested-branch example that is now the standard textbook illustration of this mechanism, following Fung et al. [8]: a SIMT group of four lanes executes the graph in Figure 10, where node labels carry the active mask of the lanes passing through them (`1` = active), and the outer branch at A and inner branch at B both diverge.

Table 1 traces the reconvergence stack (top-of-stack, TOS, listed first in each snapshot) as the SIMT group executes one loop iteration of Figure 10. Each stack entry records where to resume ("Next PC"), which lanes are active there, and the PC at which this entry's lanes should eventually reconverge with the rest of the SIMT group.

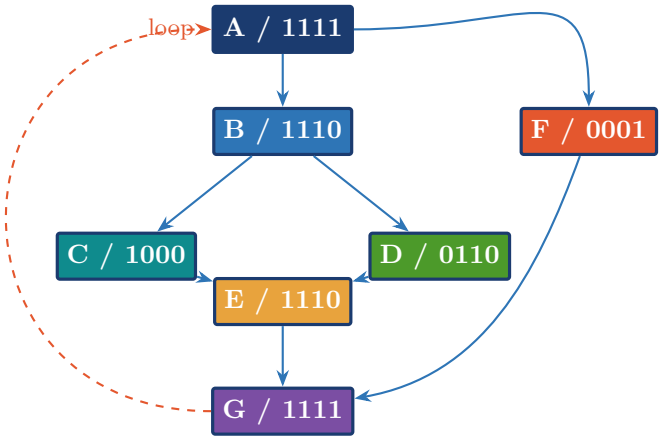


Figure 10: The classic nested-divergent-branch example (after [8]): A diverges into a 3-lane path (via B) and a 1-lane path (via F), reconverging at G; within the 3-lane path, B itself diverges into C (1 lane) and D (2 lanes), reconverging at $E = \text{IPDOM}(B)$. G loops back to A.

Stage	Stack	Reconv. PC	Next PC	Active mask
(i)	**TOS**	G	**B**	1110
		G	F	0001
	bottom	–	G	1111
(ii)	**TOS**	E	**C**	1000
		E	D	0110
		G	F	0001
	bottom	–	G	1111
(iii)	**TOS**	G	**E**	1110
	bottom	–	G	1111

Table 1: Reconvergence-stack contents while executing Figure 10, snapshotted immediately after each divergent branch: (i) just after A diverges, (ii) just after B also diverges, and (iii) once C and D have both drained into the inner reconvergence point E. Each stack is listed top-of-stack (TOS, highlighted – executed next) down to the bottom entry; popping an entry is exactly what re-merges lanes.

Reading the stack top-to-bottom gives the resulting serialized schedule: after A diverges, B's 3-lane path runs first, with F's lane masked off for the duration; within it, C (1 lane) executes, then D (2 lanes), which pops back to the merged E entry (3 lanes, the inner reconvergence at $\text{IPDOM}(B) = E$); only then does F's lone lane get to run, popping back to the outer, fully reconverged G entry (4 lanes, at $\text{IPDOM}(A) = G$). Figure 11 draws this out as a timeline: the shaded region is where the SIMT group is running below full occupancy – the direct, measurable cost of branch divergence.

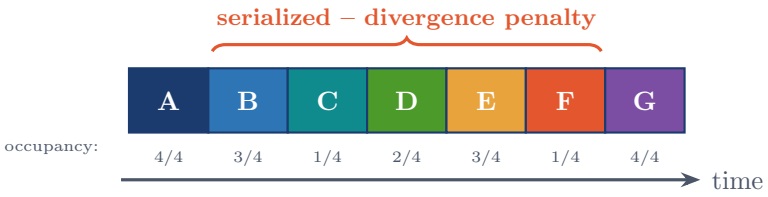


Figure 11: Execution timeline for Figure 10. A and G are the only two stages where the SIMT group runs at full, 4-lane occupancy. Every stage in between has at least one lane masked off: B (3/4) and F (1/4) are A's two divergent paths, run one after the other; and within B's path, C (1/4) and D (2/4) are B's own two divergent paths, likewise run one after the other before reconverging at E (3/4). Each reduced-occupancy stage is cycles the group spends with idle lanes – a direct, measurable cost of divergence that cannot be recovered once the branch has diverged.

Observation: The Cost Is Structural, Not Incidental

Nothing about this schedule is a missed optimization opportunity for the hardware to fix later: C, D, and F are executed *at all* only because their lanes disagreed at a branch. The reconvergence stack is already doing the best it can given a shared program counter – the real levers are (1) not diverging in the first place (data layout, thread grouping), and (2) making sure the *compiler*-visible control-flow graph is shaped so this stack discipline stays cheap, which is exactly where Sections 5 and 6 come in.

Predication as a Degenerate Case

Not every `if` needs the reconvergence machinery just described. Very short, branchless conditionals are routinely compiled with **predication** (a `select`/`cmov` instruction, executed unconditionally by all lanes, with divergent lanes' results simply discarded) rather than an actual control-flow branch. The worked example of Section 3.4 – a single `phi` merging two trivial constants – is exactly the kind of code an `if`-conversion pass would predicate away in practice; we kept it as an explicit branch there purely for illustration, and revisit real `if`-conversion versus linearization in Section 5.

4.3 Dynamic Warp Formation and Thread Block Compaction

The reconvergence stack above is a per-SIMT-group mechanism: it never looks outside the one SIMT group it belongs to. Fung et al. observed that this leaves performance on the table whenever *different* SIMT groups diverge onto the *same* path and end up, briefly, executing the same instruction at the same PC while sitting in different (partially occupied) SIMT groups. Their **Dynamic Warp Formation** (DWF) [8, 9] scheme opportunistically regroups such lanes, at run time, into new, fuller SIMT groups sharing a common PC – trading strict per-SIMT-group identity for higher SIMD utilization. A follow-up design, **Thread Block Compaction** (TBC) [7], achieves a similar effect with lower hardware complexity by compacting divergent threads at the thread-*block* granularity

using a shared reconvergence mechanism across all SIMT groups of a block, avoiding some of DWF's scheduling and memory-coalescing pitfalls. As of this writing, no successor scheme has displaced these two as the reference points for SIMT-group-level divergence mitigation in the architecture literature; more recent hardware generations instead relax the reconvergence discipline itself (next section).

4.4 Beyond the Stack: Independent Thread Scheduling

More recent hardware generations have begun relaxing the strict "all diverged lanes must fully drain before reconvergence" discipline of the classical stack, giving each thread its own program counter and call stack and allowing *interleaved* execution of diverged paths, with explicit reconvergence only at programmer- or compiler-inserted synchronization points. This removes some SIMT-induced correctness hazards of the strict stack discipline (notably certain spin-lock patterns that could deadlock under it) at the cost of requiring new compiler and language support to reason about *when* reconvergence actually happens; Abaie Shoushtary et al. [1] give a recent, detailed characterization of this design space and propose a unified abstraction ("Hanoi") that models both stack-based and independent-thread-scheduling hardware. We do not otherwise dwell on this newer hardware in this paper: the *compiler-side* problem – deciding statically which values and branches are safe to treat as uniform – is essentially unchanged, and if anything more important, because the compiler can no longer lean quite so hard on the hardware always doing the right thing for it.

4.5 When the Hardware Mechanism Breaks Down

The entire reconvergence-stack argument of Section 4.2 rests on one assumption: that $\text{IPDOM}(Q)$ *exists* and is a useful point to reconverge at. Sections 2.1 and 2.3 tell us exactly when that assumption can fail.

Observation: Failure Modes

- **Degenerate IPDOM.** If the diverged block's only guaranteed common successor is the function's single `exit` block (which happens whenever the diverged paths are riddled with early returns, unstructured jumps, or exception-like control flow), the hardware has no useful reconvergence point short of the very end of the kernel – lanes stay serialized for the rest of the function, even though a tighter, semantically safe reconvergence point may exist and simply is not syntactically visible as an IPDOM.
- **Irreducible loops.** If a diverged branch feeds into an irreducible loop region (Section 2.3) – one enterable from two or more distinct headers – the very notion of "the" loop header that the stack discipline reconverges at each iteration around becomes ill-defined. Early SIMT hardware and early compiler passes that assumed reducibility could, in the worst case, produce *incorrect* results here (lanes reconverging at the wrong point, or never reconverging), not merely slow ones – which is precisely why every serious implementation, hardware or compiler-analysis, treats irreducible control flow as a case requiring dedicated, conservative handling rather than an edge case to ignore (Section 7).

Takeaway: Section Summary

Unstructured *or* irreducible control flow in a GPGPU kernel degrades the effectiveness of hardware reconvergence, and in the worst case – when both the compiler and the hardware

are unprepared for it – can compile or execute the kernel incorrectly. This motivates the first line of compiler defense taken up next: reshaping the control-flow graph itself so that useful IPDOMs exist and loops are reducible.

5 Compiler Mitigation I: Reshaping the Control-Flow Graph

If unstructured and irreducible control flow degrade or break hardware reconvergence (Section 4.5), the first compiler-side idea worth trying is the obvious one: *stop producing unstructured or irreducible control flow*. This section and the next develop this idea, and the complementary idea of divergence analysis (Section 6), purely as general compiler theory – independent of any one compiler's implementation. This section covers two distinct, commonly conflated transformations that pursue the CFG-repair idea: (1) **node splitting** (Section 5.1), which repairs *reducibility* by turning an irreducible CFG into an equivalent reducible one; and (2) **control-flow linearization** (Section 5.2), which repairs *structuredness* by turning an unstructured CFG into an explicitly sequential, predicated one. Section 7 returns to both ideas in their production form, covering LLVM's `FixIrreducible` and `StructurizeCFGPass` passes.

5.1 Repairing Irreducibility: Node Splitting, and Why It Is Expensive

For irreducible loops specifically (Section 2.3), the classical repair is **node splitting**: cloning a node that is reachable from multiple loop headers so that each clone has a single incoming loop header, breaking the multi-entry structure that makes the loop irreducible – in other words, turning an irreducible CFG into an *equivalent*, strictly larger, reducible one.

Figure 12 shows the minimal instance of the problem: two blocks, b_1 and b_2, that branch to each other, entered from two separate, incomparable blocks p_1 and p_2 (the same shape as Figure 5(b)). Neither b_1 nor b_2 dominates the other, so neither can serve as the region's header – it contains a cycle but is not a *natural* loop (Section 2.3.1). Node splitting resolves this by cloning the entire cyclic region once per extra entry, so that each copy is reachable from only one predecessor and therefore has a well-defined, single header: the copy reachable from p_1 becomes a natural loop headed by b_1, and a fresh copy reachable from p_2 becomes a separate natural loop headed by b_2'.

Janssen and Corporaal [15] give a controlled node-splitting algorithm that minimizes the number of clones needed for larger, more tangled instances of this pattern than the minimal example above. Unfortunately, this class of technique runs into a hard theoretical wall:

Theorem: Exponential Blow-up

There exist irreducible flow graphs on n nodes for which *any* equivalent reducible flow graph obtained by node splitting must have at least 2^{n-1} nodes [5]. The intuition, long suspected as "folklore" and proven rigorously by Carter, Ferrante, and Thomborson, is that reducible graphs correspond to a strictly less expressive class of computation structures (closely related to nested, tree-like decompositions), and some irreducible graphs are combinatorially "as complex as" the complete graph on n nodes, which is a known worst case for this bound.

In other words, general node-splitting-based reducibility repair is not merely NP-complete in the worst case – it can be information-theoretically *infeasible* to apply exhaustively, which is exactly why production compilers do not attempt full reducibility repair as a general-purpose pass, and instead either (a) restrict node splitting to small, local irreducible regions where the blow-up is

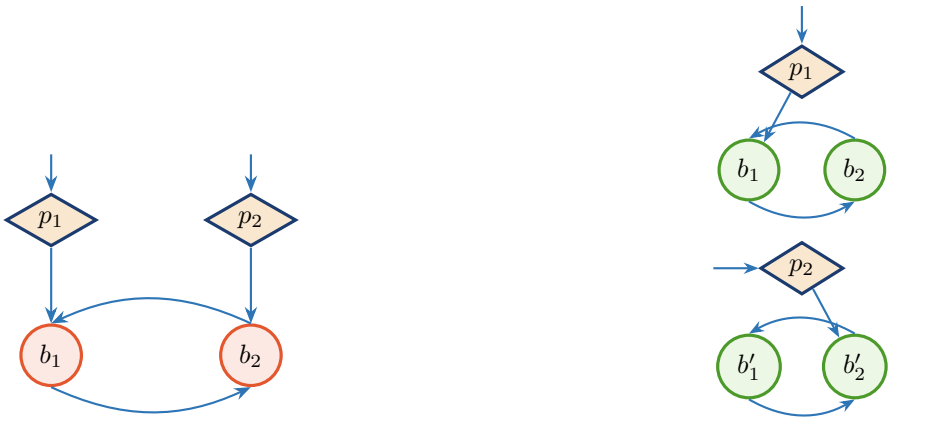


Figure 12: Node splitting applied to a minimal irreducible loop. Before: b_1 and b_2 branch to each other but are entered from two incomparable blocks p_1 and p_2, so neither can serve as a header. After: the cyclic region is cloned once per extra entry – $\{b_1, b_2\}$ becomes a natural loop headed by b_1 (entered only from p_1), and a fresh copy $\{b'_1, b'_2\}$ becomes a separate natural loop headed by b'_2 (entered only from p_2) – at the cost of doubling the region's code.

manageable, or (b), as we will see repeatedly in Section 7, build analyses that handle irreducibility *directly*, without ever attempting to eliminate it.

5.2 Repairing Unstructuredness: Control-Flow Linearization

Unstructured-but-reducible control flow is a different problem, and node splitting is not even the right tool for it: no loop is involved at all, just an ordinary acyclic region where some block has more than one way in from genuinely different branches (Section 2.2), so its IPDOM-based reconvergence point is not where the hardware needs it to be.

Following Anantpur and Govindarajan's own motivating example [4], take the short-circuited condition `if ((c1() || c2()) && c3()) S1; else S2; S3;`. Short-circuit evaluation of `||` and `&&` does not lower this to a tidy nest of if-then-elses; it lowers it directly to the six-block, branch-per-block form of Listing 2, whose labels b_1–b_6 are chosen to line up, block for block, with Figure 13.

Listing 2: The short-circuited condition `if ((c1()||c2())&&c3()) S1; else S2; S3;`, lowered to its six-block branch form. Block labels match Figure 13 one for one. Braces mark each branch's own lexical arm; `b3:` and `b5:` sit *outside every brace*, each reached by `goto`s from more than one arm – the SESE violation analyzed below.

```
b1: if (c1()) {
        goto b3;          // c1() true: short-circuits c2() entirely
    } else {
b2:     if (c2()) {
            goto b3;      // c2() true: same b3 -- crosses out
        } else {
            goto b5;      // c1()||c2() false -- crosses out
        }
    }
b3: if (c3()) {
        goto b4;          // c1()||c2() already true: c3() alone decides
    } else {
```

```
        goto b5;          // same b5 -- crosses out
    }
b4: S1;          goto b6;
b5: S2;          goto b6;
b6: S3;
```

The listing's braces make this a lexical fact, not just a graph-theoretic one, and Section 2.2's SESE definition pins down exactly where it fails – in two separate places, each breaking a different half of the definition. First, take the region $R_1 = \{b_1, b_2\}$ – the code that lowers the short-circuited sub-expression `c1()||c2()`, i.e. everything inside b_1's own braces. R_1 is connected (the $b_1 \to b_2$ edge) and has a single entry (from before b_1), so it is a candidate SESE region, and Section 2.2 requires it to also have *exactly one* edge leaving it. It has three: $b_1 \to b_3$, $b_2 \to b_3$, and $b_2 \to b_5$ – all three visible in Listing 2 as `goto`s that cross back out past b_1's or b_2's closing brace. R_1 therefore fails the single-*exit* half of the SESE definition, and this is exactly the mechanical fingerprint of source (ii) in Section 2.2's list of unstructured-control-flow culprits: a single boolean computed once and branched on *once* would give R_1 a clean, two-edge exit (one per truth value), but short-circuit evaluation instead jumps straight to the eventual continuation from *every* point the answer becomes known, so b_1's own direct edge to b_3 competes with b_2's.

Second, and independently, take the single-block region $R_2 = \{b_3\}$. On its own terms R_2 has a perfectly well-defined exit – its own `c3()` test, branching to b_4 and b_5 – so the exit side is not the problem here. The entry side is: two edges reach R_2 from outside it, $b_1 \to b_3$ directly and $b_2 \to b_3$, so R_2 fails the single-*entry* half of the same definition. b_5 fails the identical way, as its own region $R_3 = \{b_5\}$: it too is entered by two edges, $b_2 \to b_5$ and $b_3 \to b_5$. R_1, R_2, and R_3 are three separate candidate SESE regions, each failing its *own* half of the definition – R_1 on exit, R_2 and R_3 on entry – and it is not a coincidence that R_1's three stray exits are exactly the edges landing as R_2's and R_3's stray entries: that is simply what it looks like when one region's boundary is crossed by edges that immediately become another region's boundary too.

To be precise about what actually breaks here: it is *not* that the hardware is somehow incapable of reconverging at the IPDOM. b_6 (`S3`, the code after the whole `if`) is the true IPDOM of the whole construct, and Table 3 confirms it is reconverged correctly, exactly once, with all lanes present (stage 8, mask 1111). b_3's problem is that it sits strictly *between* the divergent branch at b_1 and that true IPDOM, and the reconvergence-stack discipline of Section 4.2 only merges active masks that arrive at a designated reconvergence point *at the same stack depth, at the same time* – a role b_3 never holds for either path that reaches it. So each incoming path to b_3 keeps its own, separate stack entry, and b_3 is visited once per incoming path rather than once, total: for a SIMT group where some lanes take the direct edge from b_1 and others arrive via b_2, b_3's instructions execute twice – exactly the redundant-execution pathology that Wu et al. [35] document empirically in bulk-synchronous GPU kernels. b_5 fails the identical way, for the same reason. The structural defect, in short, is not a failure to reconverge at the IPDOM – it is *redundant re-execution* of the merge blocks that lie strictly between a divergent branch and its true IPDOM.

If the compiler can determine, statically, that these branch conditions are divergent (Section 6) but that no cheaper restructuring is available, **control-flow linearization** repairs this by turning the unstructured region into a structured, explicitly sequential one: every reachable block is laid out exactly once, in a single fixed order, each instruction guarded by the lane predicate under which the original code would have reached it, so that b_3 and b_5 each appear exactly once in the linearized instruction stream (Figure 14).

To make the benefit of linearization concrete and countable, rather than purely qualitative, Table 2 fixes one specific 4-lane instance of Figure 13/Listing 2 – a particular assignment of `c1()`,

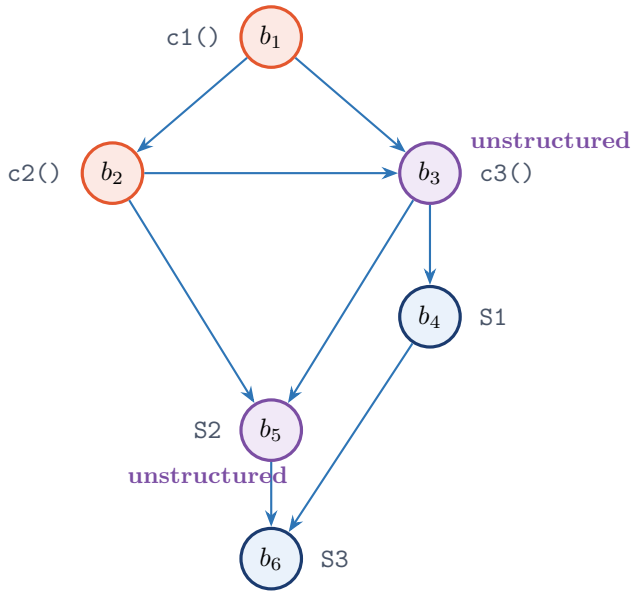


Figure 13: Before linearization: control flow for the source of Listing 2. b_1–b_3 test `c1()`, `c2()`, `c3()`; b_3 (reached from both b_1 and b_2) and b_5 (reached from both b_2 and b_3) are unstructured merges that sit before the region's true IPDOM, b_6 – so a naive reconvergence-stack schedule visits each of them once per incoming path rather than once, total.

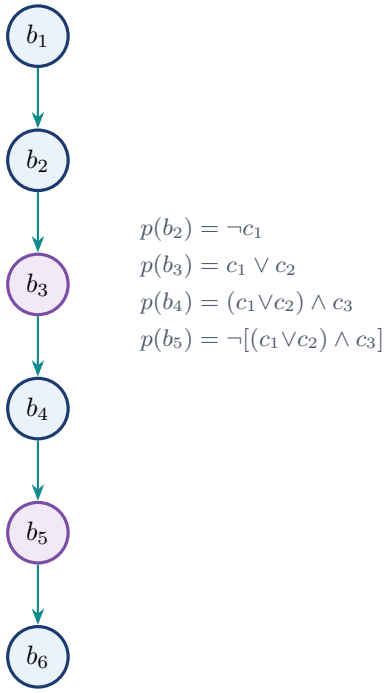


Figure 14: After linearization: all six blocks are laid out exactly once, in a single fixed order, each guarded by the lane predicate under which the original code would have reached it (b_1 and b_6 are unconditional). b_3 and b_5 – previously visited once per incoming path – now each have a single static occurrence, executed once under the disjunction of both paths' original predicates.

`c2()`, `c3()` per lane – and Tables 3 and 4 simulate its execution both ways: once under the classical reconvergence-stack discipline of Section 4.2 applied directly to the unstructured CFG, and once after linearization.

Lane	`c1()`	`c2()`	`c3()`	Path taken
1	T	–	T	$b_1 \to b_3 \to b_4$ (`S1`)
2	T	–	T	$b_1 \to b_3 \to b_4$ (`S1`)
3	F	T	F	$b_1 \to b_2 \to b_3 \to b_5$ (`S2`)
4	F	F	–	$b_1 \to b_2 \to b_5$ (`S2`)

Table 2: A concrete 4-lane assignment of `c1()`/`c2()`/`c3()` for Figure 13, used to simulate execution both before (Table 3) and after (Table 4) linearization. A "–" marks a condition that short-circuit evaluation never evaluates for that lane.

Stage	Block	Mask	Occ.	Note
1	b_1	1111	4/4	diverges: lanes 1,2 direct to b_3; lanes 3,4 to b_2
2	b_3	1100	2/4	1st visit – direct arrival (lanes 1,2)
3	b_4	1100	2/4	lanes 1,2 reach `S1`; hold, waiting at b_6
4	b_2	0011	2/4	diverges: lane 3 to b_3; lane 4 to b_5
5	b_3	0010	1/4	2nd visit – via-b_2 arrival (lane 3)
6	b_5	0010	1/4	1st visit – via-b_3 arrival (lane 3)
7	b_5	0001	1/4	2nd visit – direct-b_2 arrival (lane 4)
8	b_6	1111	4/4	all four lanes finally reconverge

Table 3: Execution order for Table 2 under the classical reconvergence-stack discipline, applied directly to the unstructured CFG of Figure 13. b_1 and b_2 are genuine divergent branches: each pushes two stack entries that both correctly reconverge at the true IPDOM, b_6. But b_3 and b_5 are not themselves the IPDOM of anything – each of their two incoming edges gets pushed as an independent stack entry, so each executes *twice* (highlighted), at reduced occupancy both times, instead of once at the occupancy of their true, combined lane set.

Comparing the two traces makes the benefit countable, not merely qualitative: the same four lanes, computing the same six blocks' worth of work, cost **8** dynamic block-visits under the reconvergence-stack discipline (Table 3) – two of them, at b_3 and b_5, pure repeats of a block already executed for other lanes – versus exactly **6** after linearization (Table 4), one per block, no repeats. Nor is this merely a reshuffling of the same cost: linearization never raises a block's occupancy above what its own combined lane set already required (b_3 goes from two visits at 2/4 and 1/4 occupancy to one visit at 3/4; b_5 from two visits at 1/4 and 1/4 to one visit at 2/4), so the two eliminated visits are a strict reduction in total dynamic work for this example. The price, visible in Table 4 itself, is that every lane now occupies every block's slot in program order: b_4's occupancy (2/4) is unchanged, but lanes 3 and 4 – which never reach b_4 at all under the original code – must still sit through that instruction slot with their predicate off, contributing no work yet costing an issue cycle they would not have spent had the branch stayed an ordinary, unstructured one.

Stage	Block	Mask	Occ.	Note
1	b_1	1111	4/4	unconditional
2	b_2	0011	2/4	$p(b_2) = \neg c_1$
3	b_3	1110	3/4	$p(b_3) = c_1 \vee c_2$ – one merged visit
4	b_4	1100	2/4	$p(b_4) = (c_1 \vee c_2) \wedge c_3$
5	b_5	0011	2/4	$p(b_5) = \neg[(c_1 \vee c_2) \wedge c_3]$ – one merged visit
6	b_6	1111	4/4	unconditional

Table 4: Execution order for the same lane assignment (Table 2) after linearization (Figure 14). Every block now executes exactly once, in program order, each lane's contribution gated by its predicate. b_3 and b_5 – fragmented into two reduced-occupancy visits each in Table 3 – are each computed once (highlighted), at the occupancy of their full, combined lane set.

Observation: Not the Same as If-Conversion

It is tempting to conflate control-flow linearization with classical *if-conversion* (predication) – and the worked example above does nothing to dispel that: Figure 14 and Table 4 are exactly what fully if-converting the whole b_1–b_6 nest would also produce, branch for branch. That is not a coincidence: whenever the minimal region that needs repairing turns out to be the *entire* divergent branch nest, as it does here, full linearization and full if-conversion of that nest coincide, mechanically. The two techniques differ in *why* and *how much* of the CFG each one touches, a distinction this small, fully-divergent example is too minimal to show. If-conversion is a per-branch cost bet, orthogonal to structuredness: it asks whether a branch's bodies are short and *uniform enough* that the wasted work on masked-off lanes is cheaper than keeping the branch, and that bet applies just as well to a branch that is already perfectly structured – if-converting a clean, single-merge if-then-else is a performance choice, not a correctness-adjacent one. Linearization, in the sense used by Anantpur and Govindarajan [4] and by Moll and Hack [24], is triggered by a structural question instead: does this region satisfy SESE (Section 2.2), so the reconvergence-stack hardware can find one true IPDOM? A region that fails this test needs *some* repair regardless of body size or how likely its lanes are to agree, because the alternative is the redundant re-execution demonstrated above, not merely a missed optimization. Moll and Hack's **partial** control-flow linearization makes the difference in *scope* concrete: it serializes only the minimal sub-region that a dominance/post-dominance analysis identifies as unstructured, and leaves every already-structured branch elsewhere in the function – however large, and whether or not it is divergent – as a real branch. A cost-driven if-conversion heuristic has no such structural target: applied on its own, it can leave an unstructured region entirely unrepaired if its bodies look too large to bet on, or predicate a branch that was already structured and never needed repair at all.

Takeaway: Section Summary

Structural CFG repair – node splitting for reducibility, control-flow linearization for structuredness – treats every branch uniformly and pays a real, often substantial, performance cost wherever it applies (redundant predicated work on masked lanes, loss of branch prediction and early-exit opportunities). It is the right tool when a region's control flow is *genuinely* pathological for the hardware. But most branches in most kernels are perfectly well-structured and *not* divergent at all – and linearizing or restructuring those branches is pure waste. What is

missing is a way to know, statically, *which* branches (and which values) are actually divergent – the subject of Section 6. Section 7 shows how both of these general ideas, CFG repair and divergence analysis, were carried into production inside a single real compiler.

6 Compiler Mitigation II: Divergence Analysis

6.1 Divergent and Uniform Variables

Definition: Divergent / Uniform Variable

Let `Var` be a variable (or, more generally, an SSA value) computed at some program point by every lane of a SIMT group. If, for some input, `Var` takes on *at least two different values* across the group's active lanes, `Var` is **divergent** (equivalently, **varying**). Otherwise – if `Var` always takes the same value on every active lane – `Var` is **non-divergent**, equivalently **uniform** or **convergent**.

Divergent variables are the root cause of two distinct performance phenomena:

- **Branch divergence** (Section 4): when a branch's condition variable is divergent, different lanes disagree on the branch outcome, triggering the serialization machinery of Section 4.2.
- **Memory divergence**: when an address (pointer) variable used in a load or store is divergent, different lanes access different, potentially far-apart memory addresses in the same instruction. Unlike a uniform, coalesced access (one cache line serves the whole SIMT group), a divergent address pattern can produce one cache miss per distinct address, and the entire group must stall until the *last* lane's access is satisfied.

6.2 Motivation: Two Worked Examples

Example 1: Selective linearization.

Blanket control-flow linearization (Section 5) rewrites every branch it touches, regardless of whether that branch is actually divergent. If the compiler instead *knows* – via a sound static analysis – that a given branch's condition is uniform, it can skip linearizing that branch entirely and emit it as an ordinary, cheap control-flow branch, applying the (expensive) restructuring transformation only where it is load-bearing. This is not merely hypothetical: Section 7 describes a production pass that exposes exactly this kind of divergence-analysis-guided opt-out.

Example 2: Scalarization.

Consider the vector instruction of Listing 3, executed identically by every lane of a SIMT group.

Listing 3: A vector add, a candidate for scalarization.

```
1 %vdest = vadd %vsrc1, %vsrc2
```

If divergence analysis determines that `%vsrc1` and `%vsrc2` are both uniform (hence `%vdest` is uniform too), the compiler can replace the vector operation with a single *scalar* add, computed once and broadcast to all lanes, rather than redundantly computing the identical result once per lane.

Both examples share the same shape: a transformation that is always *correct* to apply universally, but only *profitable* to apply selectively – and profitability hinges entirely on a piece of information (uniform vs. divergent) that does not exist anywhere in the unanalyzed IR. Divergence analysis exists to manufacture that information.

6.3 Problem Statement

Problem: Divergence Analysis (DA)

Given a control-flow graph (G, s) of a SIMT kernel P:

- compute the set of divergent variables at each program point of G (equivalently, dually, the set of uniform variables); or
- classify every instruction I as **uniform** or **divergent**, depending on whether I produces the same result on every active SIMT lane or not.

The practical difficulty of this problem is governed by several largely independent axes:

- the **hardware reconvergence mechanism** assumed (stack-based IPDOM reconvergence versus dynamic warp formation versus independent thread scheduling change what "divergent" even implies operationally, cf. Section 4);
- the **shape of the underlying CFG** – reducible versus irreducible (Section 2.3), structured versus unstructured (Section 2.2);
- the **granularity of program points** considered – per statement, per basic block, or whole-function summaries;
- the **precision/complexity trade-off** of the underlying algorithm (graph reachability versus constraint solving versus abstract interpretation, Section 6.5); and
- the **implementation substrate** (a research prototype versus a production compiler's IR and pass-manager constraints, Section 7).

Despite this variety, essentially every divergence analysis in the literature and in practice reduces to some flavor of **fixpoint worklist algorithm** over a two-point lattice $\{\mathsf{uniform} \sqsubset \mathsf{divergent}\}$: start from a small set of *known-divergent* sources, and propagate divergence forward through the program's dependence structure until no more values can be marked divergent.

6.4 A Simple Worklist Algorithm

The propagation step needs two distinct kinds of dependence edges, not just one:

- **Data dependence**: the ordinary use-def relationship. If `x` is divergent and `y = f(x, ...)`, then `y` is divergent (unless f is specifically known to be a uniformity-restoring operation, such as a SIMT-group vote reduction).
- **Sync dependence**: the control-dependence-turned-data-flow relationship of Allen et al. [3] (Section 2.4), specialized to SIMT. A value `v` is **sync-dependent** on a divergent branch B if `v` is defined by a `phi` node at (or reachable from) B's reconvergence point, and the incoming value selected by that `phi` differs depending on which side of B was taken.

Listing 4 makes sync-dependence concrete, revisiting the running example of Listing 1: although `%a` is not on `%tid`'s use-def chain (it is not *data*-dependent on `%tid`), `%a` *is* sync-dependent on `%tid`, because which of the `phi`'s two incoming constants is selected is entirely determined by `%cond`, which is data-dependent on `%tid`.

Listing 4: Sync-dependence: `%a` is not data-dependent on `%tid`, yet must be marked divergent.

```
%cond = icmp slt i32 %tid, 10
br i1 %cond, label %then, label %else
then:
  br label %merge
else:
  br label %merge
merge:
  %a = phi i32 [ 0, %then ], [ 1, %else ]   ; sync-dependent on %tid
```

Algorithm 1 presents a simple, deliberately unsophisticated worklist algorithm capturing both kinds of propagation. It is instructive precisely because it is naive: it is neither sound (it can undermark divergence in the presence of loops with divergent trip counts feeding back into supposedly-uniform values) nor precise (it over-approximates sync-dependence conservatively) nor guaranteed to terminate usefully on a complicated, irreducible CFG. Every algorithm surveyed in Sections 6.5 and 7 can be understood as a progressively more careful, more sound, and/or more precise refinement of exactly this skeleton.

Algorithm 1: DivergenceAnalysis (simple worklist form)

Input: (G, s) – control-flow graph G with start node s

Output: *DAVars* – the set of divergent variables within G

Method:

1: $DAVars \leftarrow \emptyset$
2: Initialize *Worklist* with the *trivial* divergent variables of G (e.g. the lane/thread-ID intrinsic)
3: **while** $Worklist \neq \emptyset$ **do**
4: $\quad v \leftarrow Worklist.\textsc{Pop}$
5: $\quad DAVars \leftarrow DAVars \cup \{v\}$
6: $\quad$ **for each** variable u **data-dependent** on v, with $u \notin DAVars$ **do**
7: $\quad\quad Worklist.\textsc{Push}(u)$
8: $\quad$ **end for**
9: $\quad$ **for each** variable u **sync-dependent** on v, with $u \notin DAVars$ **do**
10: $\quad\quad Worklist.\textsc{Push}(u)$
11: $\quad$ **end for**
12: **end while**
13: **return** *DAVars*

6.5 A Survey of the Divergence-Analysis Literature

6.5.1 Coutinho et al. (2011): Formalizing Divergence via GSA

Coutinho, Sampaio, Pereira, and Meira's *Divergence Analysis and Optimizations* [6] is the first formal treatment of the problem. They define a small formal assembly language, **μ-SIMD**, and represent it in **Gated Static Single Assignment** (GSA) form – an extension of SSA in which `phi` nodes carry explicit *predicate* operands recording which incoming control-flow edge was actually taken. Representing the program in GSA form is precisely a mechanism for converting sync-dependence into ordinary data-dependence (an explicit predicate value becomes just another SSA operand, cf. Section 2.4): a full **data-dependence graph** is then built over the GSA-form program, and divergent variables are computed by a straightforward **graph-reachability** query from the trivially-divergent sources – structurally the algorithm of Algorithm 1, made precise and given a soundness argument relative to the GSA construction.

6.5.2 Sampaio et al. (2014): A Constraint-Based Reformulation

The journal-length follow-up, *Divergence Analysis* [32], keeps the same GSA-based problem formulation but replaces graph reachability with a **constraint system**: each program construct emits a small set of Boolean/set constraints relating the divergence status of its operands and result, and the least fixed point of the resulting constraint system is computed directly. The authors argue – and give proofs to substantiate – that this reformulation is both **sounder** (certain reachability-based false negatives around loop-carried divergence are eliminated) and **more precise** (fewer false positives) than the original 2011 graph-reachability formulation, while remaining implementable as a standard iterative dataflow analysis.

6.5.3 Karrenberg and Hack (2012): A Forward Dataflow Formulation for CPU SIMD

Improving Performance of OpenCL on CPUs [18], building on the same group's **Whole-Function Vectorization** [17] work, targets a different but closely related backend: compiling OpenCL kernels to SIMD CPU instructions rather than GPU SIMT hardware. Their goal is explicitly the same as Example 1 of Section 6.2: identify divergent branches so that the whole-function vectorizer's control-flow linearization (there, an explicit if-conversion pass over the vectorized code) can be **restricted** to only the branches that actually need it. They model the problem as an ordinary **forward dataflow analysis** over the CFG – uniformity information for a value is computed directly from the uniformity of its operands via a monotone transfer function per instruction, iterated to a fixed point in the standard Kildall style – which trades some of the GSA formulations' conceptual elegance for a formulation that maps directly onto existing dataflow-analysis infrastructure in a production-style compiler.

6.5.4 Rosemann, Moll, and Hack (2021): An Abstract Interpretation for Reducible CFGs

An Abstract Interpretation for SPMD Divergence on Reducible Control Flow Graphs [27] steps back from algorithm-first design and asks a semantic question first: *what, exactly, should "uniform" mean, formally, for a value at a given program point?* The authors give this meaning via **trace semantics**: a value is uniform at a point if, for every pair of threads that reach that point along traces satisfying an appropriate control-flow correspondence, the value agrees. They then derive a sound **abstract interpretation** of this semantics – an abstraction that is provably conservative with respect to the trace-semantic definition – and show that, restricted to **reducible** control-flow graphs, it can be computed efficiently using loop-nesting-forest structure (natural loop headers

double as the points at which divergence introduced inside a loop must be assumed to persist until the loop's unique exit reconvergence). This paper is the direct theoretical ancestor of LLVM's current Uniformity Analysis (Section 7.2.4), and its restriction to reducible graphs is exactly the gap that Sahasrabuddhe et al. close next.

6.5.5 Sahasrabuddhe et al. (2022): Extending to Irreducible CFGs

Uniformity Analysis for Irreducible CFGs [30, 31] generalizes Rosemann et al.'s abstract interpretation from natural loops to the **cycle** abstraction of Havlak [12] (Sections 2.3 and 2.4), which admits multiple entries per cycle and therefore covers irreducible loop nests as a strict generalization rather than a special case requiring separate machinery. We defer the technical heart of this paper – the notion of **maximal convergence** and m**-converged** static instances – to Section 7.2.4, since it is presented there in tandem with its LLVM implementation, with which it was co-developed.

6.5.6 Related Work: Divergence-Aware Control-Flow Melding

Finally, it is worth situating divergence analysis relative to a complementary, more aggressive line of work: Saumya, Sundararajah, and Kulkarni's **DARM** [33] does not merely *classify* divergence – it uses divergence information to actively **meld** (merge) syntactically different but semantically similar divergent control-flow paths into a single, shared instruction sequence with per-lane data selection, going a step further than either linearization (Section 5) or a scalarization-only consumer of divergence analysis (Section 6.2). It is a useful reminder that divergence analysis, as covered in this paper, is best understood as *infrastructure*: the analyses surveyed above are valuable largely in proportion to the optimizations built on top of them, of which control-flow melding is one of the more sophisticated recent examples.

Takeaway: Section Summary

Divergence analysis has been reinvented, from partially independent motivations, at least three times in the academic literature (GSA reachability, constraint solving, forward dataflow) before converging, in Rosemann et al.'s and Sahasrabuddhe et al.'s work, on a semantically grounded abstract-interpretation formulation general enough to handle irreducible control flow soundly. The next section traces how this same arc played out, largely independently and on a slower clock, inside a single production compiler: LLVM.

7 Compiler Mitigation in Production: A Patch-Level History Inside LLVM

Sections 5 and 6 developed two lines of compiler mitigation – structural CFG repair (node splitting and control-flow linearization) and divergence analysis – purely as general compiler theory, deliberately independent of any one compiler's internals. This section grounds both lines of work in a single production compiler, LLVM, at the level of individual patches and differential revisions. Section 7.1 covers LLVM's two structural CFG-repair passes: `FixIrreducible`, the production counterpart to Section 5.1's node splitting, and `StructurizeCFGPass`, the production counterpart to Section 5.2's control-flow linearization – including how the latter can consult a divergence analysis to restructure selectively rather than unconditionally, precisely the Example 1 scenario of Section 6.2. Section 7.2 then turns to divergence analysis itself, and to why it needed *three* successive from-scratch implementations to get right.

7.1 LLVM's Control-Flow Repair Passes

Take structural CFG repair first: the two ideas developed as pure compiler theory in Section 5 – node splitting for reducibility and control-flow linearization for structuredness – each have a concrete production counterpart inside LLVM, `FixIrreducible` and `StructurizeCFGPass`, taken up in turn below.

7.1.1 `FixIrreducible`: Production Node Splitting, Without the Cloning

D77198, *Introduce fix-irreducible pass* [28], is LLVM's production answer to the reducibility-repair problem of Section 5.1 (**D77198** is a differential revision on Phabricator – the web-based code-review tool LLVM used for patch review before migrating to GitHub pull requests in 2023 – and every patch identifier of this `D`*nnnnn* form used from here on links the same way, to its revision page on `reviews.llvm.org`). It was authored by Sameer Sahasrabuddhe – the same engineer who would later lead the Uniformity Analysis effort of Section 7.2.4 – and landed in April 2020. It deliberately does *not* implement classical node splitting: rather than cloning the irreducible region once per extra entry and risking the exponential blow-up of the Carter–Ferrante–Thomborson bound, `FixIrreducible` (`llvm/lib/Transforms/Utils/FixIrreducible.cpp`) introduces a single new dispatch block that all of the region's external entries, and all edges incident on its original headers, are redirected through, using a shared `ControlFlowHub` utility that also migrates any affected `phi` nodes to preserve SSA form. Because every path into the region now passes through this one new block, the region becomes, by construction, a strongly connected component dominated by it – a natural loop headed by the new block, satisfying Section 2.3's reducibility criterion – without duplicating a single instruction. The transformation trades node splitting's potential code-size blow-up for a small, fixed amount of extra runtime branching through the dispatch block; it depends on `DominatorTree` and `CycleInfo` (Section 7.2.3), optionally updates `LoopInfo`, and cannot handle indirect branches (which must be lowered to ordinary conditional branches first). The patch's own description is direct about its motivation: it is "a useful workaround for a limitation in the structurizer which produces incorrect control flow in the presence of irreducible regions" – that is, an optional prerequisite the AMDGPU backend can run just before the pass described next.

7.1.2 `StructurizeCFGPass`: Production Control-Flow Linearization

LLVM's production counterpart to control-flow linearization (Section 5.2) is `StructurizeCFGPass` – `llvm/lib/Transforms/Scalar/StructurizeCFG.cpp` (unlike `FixIrreducible` above, no single, precisely-dated patch identifier is cited for its origin here – `StructurizeCFGPass` is long-standing AMDGPU backend code that predates LLVM's Phabricator-tracked patch history entirely, and no citable revision introducing it could be substantiated for this paper). It walks the region tree of a function (conceptually the same nested-region view computed by a Program Structure Tree, Section 2.4) and rewrites each non-SESE region into an equivalent SESE one using `Flow` blocks and boolean predicate threading, so that every branch it touches becomes amenable to the standard IPDOM-based reconvergence of Section 4.2. `StructurizeCFGPass` is a hard dependency of AMDGPU kernel code generation: because the AMDGPU targets have historically relied on wave-level predicate masks and hardware constructs (`EXEC` mask push/pop) that assume genuinely structured input, the AMDGPU backend runs `StructurizeCFGPass` unconditionally before instruction selection for kernels with any non-trivial control flow. Its relationship to the academic linearization literature of Section 5.2 is, by the LLVM project's own admission in code comments and mailing-list discussion, informal – it long predates, and is not a direct implementation of, either Anantpur/Govindarajan's or Moll/Hack's published algorithms, though

it solves a closely related problem with broadly similar means (predicate threading through a restructured, SESE-respecting CFG).

By default, `StructurizeCFGPass` restructures every eligible region unconditionally – exactly the blanket, divergence-agnostic behavior motivating Example 1 of Section 6.2. It also exposes an opt-in `SkipUniformRegions` mode that does what that example asks for: it consults a divergence analysis (concretely, the `UniformityInfo` introduced in Section 7.2.4 below) to skip restructuring any region whose branches are all provably uniform, applying the restructuring transformation only where a branch is actually divergent, rather than unconditionally.

7.2 LLVM's Divergence-Analysis Implementations

The academic arc traced for divergence analysis in Section 6.5 – graph reachability, then constraint solving, then forward dataflow, then abstract interpretation restricted to reducible graphs, then generalized to irreducible graphs – is not merely a literature-review convenience. It is, almost move for move, also the history of divergence analysis *inside LLVM*, LLVM having gone through three successive from-scratch implementations between 2015 and 2022 before arriving at its current design. Figure 15 lays out the sequence of Phabricator differential revisions, the rest of this subsection covers, one at a time.

A scope note before proceeding: the revisions profiled below are the major, from-scratch implementation milestones – each one landing a new analysis or pass wholesale, as Figure 15 lays out. Every one of them also accumulated its own long tail of incremental bug fixes, refactorings, and precision improvements after landing, which are not individually covered here; the goal of this subsection is to trace how the underlying *ideas* evolved from one from-scratch implementation to the next, not to catalog every commit that subsequently touched them.

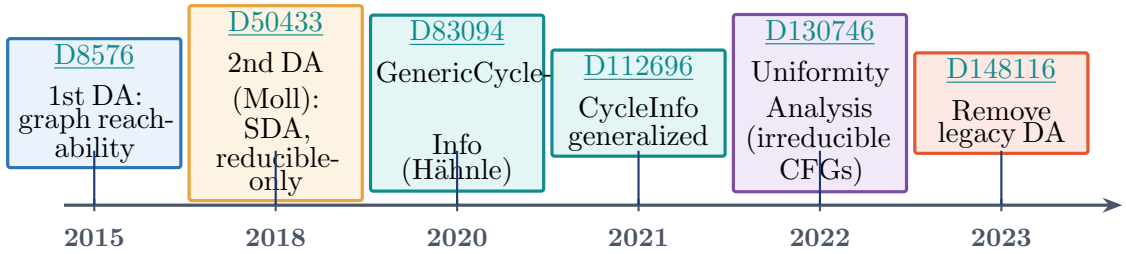


Figure 15: Timeline of divergence/uniformity analysis inside LLVM. Each patch identifier links to its Phabricator revision page.

7.2.1 First Implementation (2015): Graph-Reachability Divergence Analysis

The first divergence analysis landed in LLVM as differential revision **D8576**, *Divergence analysis for GPU programs* [36], authored by Jingyue Wu and committed in April 2015 – built and tested specifically against LLVM's NVPTX backend, per the patch's own file changes and test suite. It implements, essentially directly, the algorithm of Sampaio et al.'s *Divergence Analysis* [32] (Section 6.5): starting from trivially divergent sources (thread/lane ID intrinsics, values loaded from address spaces the target cannot guarantee are uniform), it propagates divergence forward through both data dependence (ordinary use-def chains) and sync dependence, via graph reachability over the resulting dependence graph, in a single self-contained `DivergenceAnalysis.cpp` (later, in commit `ab58c74d`, its class definition was hoisted into a header for reuse). Conservative defaults were baked in throughout: the analysis is strictly intra-procedural, treats arguments and return values of

non-kernel functions as divergent by default, and treats any load from the generic or local address space as divergent, since aliasing information about who else might be writing concurrently is not generally available at that level of the pipeline.

This implementation shipped and remained LLVM's production divergence analysis for years, but accumulated two classes of problems that eventually motivated a rewrite: it had no principled treatment of unstructured or irreducible control flow – its sync-dependence computation implicitly assumed well-behaved post-dominance relationships that simply do not hold once a CFG strays from that assumption (Section 4.5) – and it accumulated a number of correctness bugs in its sync-dependence handling over its lifetime, which is precisely the kind of code that is hard to get exhaustively right with an ad hoc, non-formalized propagation rule.

7.2.2 Second Implementation (2018): A Generic, Lazy Sync-Dependence Framework

In August 2018, Simon Moll – at the time a graduate student at Saarland University working on the **RV** (Region Vectorizer) whole-function vectorization project, the same research lineage as Karrenberg and Hack's work (Section 6.5) – proposed *A New Divergence Analysis for LLVM* on the llvm-dev mailing list, implemented as a four-patch series rooted at differential revision **D50433** [23]:

- **D50434** renamed the existing (2015) analysis to `LegacyDivergenceAnalysis`, clearing the `DivergenceAnalysis` name for the new implementation and letting both coexist during the transition.
- **D51491** introduced the two core new components: a **SyncDependenceAnalysis** (SDA) that computes join points **lazily** – rather than pre-computing full dominance-frontier information up front, it reduces the join-point problem to something structurally similar to SSA ϕ-placement, computing join points on demand as they are needed – and a generic, worklist-based **DivergenceAnalysis** (DA) that propagates values over the $\{\mathsf{uniform}, \mathsf{divergent}\}$ lattice using both data- and sync-dependence edges supplied by SDA, structurally the flow-sensitive descendant of Algorithm 1.
- **D53493** added a `GPUDivergenceAnalysis` front-end layering GPU-kernel-specific source identification (thread-ID intrinsics, atomics, etc.) on top of the generic DA/SDA core, for unstructured GPU kernels specifically.
- A fourth component, `LoopDivergenceAnalysis` aimed at vectorization use cases, was posted for reference but never committed.

Moll's own description of the design explicitly traces its lineage to RV, where a considerably richer lattice (tracking not just uniform-vs-divergent but also *stride* and *alignment* information, useful for vectorized code generation) had already been built; for the LLVM upstream patch, the lattice was deliberately simplified back down to a plain two-point set, in the interest of staying compatible with – and a drop-in, more general replacement for – the existing legacy analysis's consumers. This second implementation's signature strength was flow-sensitivity and its lazy, dominance-frontier-free join computation; its signature limitation, by the implementation's own documentation, was that it handled irreducible loops only **conservatively** – correctly, but pessimistically, marking wide swaths of an irreducible region as divergent by default rather than proving what could safely be treated as uniform. (This patch series was formally marked *Abandoned* on Phabricator in September 2022, once the third implementation described in Section 7.2.4 had fully superseded it in trunk.)

7.2.3 An Interlude: Generalizing Loops to Cycles

Precisely handling irreducible control flow requires a foundational piece of infrastructure that plain `LoopInfo` cannot provide: a notion of "loop" general enough to admit *multiple* entry points. LLVM acquired this piece in two stages, both building directly on Paul Havlak's classical algorithm for **nesting of reducible and irreducible loops** [12]:

- **D83094**, *Add a GenericCycleInfo analysis* [11], authored by Nicolai Hähnle in mid-2020, introduced `GenericCycleInfo`: cycles are computed **recursively** as a **forest** rather than a single structure spanning the whole function – the **top-level cycles** are the maximal strongly connected regions found directly by a depth-first search from the function's entry block, and each cycle's nested child cycles are the maximal strongly connected regions that remain once that cycle's header is removed – uniformly covering reducible natural loops and irreducible multi-entry regions in a single framework, in contrast to `LoopInfo`, which is only well-defined for the former.

- **D112696**, *CycleInfo: Introduce cycles as a generalization of loops* [29], authored by Sameer Sahasrabuddhe (based on Hähnle's original implementation) and landed in December 2021, generalized this into the template infrastructure LLVM ships today – `GenericCycleInfo<ContextT>` and `GenericCycleImpl<ContextT>`, parameterized so the same algorithm serves both LLVM IR and Machine IR, plus a new `GenericSSAContext` abstraction layer and a `CycleTerminology.rst` documentation file fixing the vocabulary. A **reducible cycle** is, in this framework, simply a cycle with a single entry (coinciding with the classical natural-loop notion); an **irreducible cycle** is one with multiple entries – exactly the structure that defeats `LoopInfo` and that Havlak's original algorithm was designed to characterize.

Definition: Cycle (Havlak-style, as used by `GenericCycleInfo`)

A region of a CFG is **strongly connected** if every node in it can reach every other node in it using only edges that stay inside the region. For any region of two or more nodes, this already forces a cycle to exist: if node u can reach node v and v can reach u, splicing those two paths end to end gives a closed walk, and every closed walk contains an actual cycle. The one case "strongly connected" does not settle by itself is a region consisting of a single node: graph theory counts a lone node as trivially reaching itself over a zero-length path, so by the reachability definition alone, an ordinary basic block with no back edge to itself would still count as (trivially) strongly connected, even though it plainly is not a loop. A **cycle** closes this gap by requiring, in addition, at least one **internal edge** – an edge with both endpoints inside the region, as opposed to one crossing its boundary: a maximal strongly connected region with at least one internal edge. For two or more nodes this adds nothing new (a strongly connected region of that size already has internal edges, along the very paths that make it strongly connected); for a single node it is decisive, since a self-loop is the only edge that can have both endpoints at that one block, so a lone block is a cycle only if it has one.
Cycles form a **forest**, not a single structure spanning the whole function: the **top-level cycles** are found directly by a depth-first search from the function's entry block, and each cycle C's nested **child cycles** are found the same way, one level down, in the subgraph induced by removing C's header (defined next). A cycle's **entries** are the nodes reachable from the function's entry along a path that visits no other node of the cycle – these are the ways control can get into the cycle from outside it; a cycle's **header** is whichever entry the depth-first search happens to visit first. The cycle is reducible if it has exactly one entry (which is then

unambiguously its header, since there is nothing else for the depth-first search to visit first) and irreducible if it has more than one.

This machinery is the direct enabling dependency for the third divergence-analysis implementation: with cycles available as a first-class, irreducibility-aware generalization of loops, an abstract interpretation defined in terms of "what happens at a loop header" can be restated, essentially unchanged in spirit, in terms of "what happens at a cycle header," and inherit irreducible-CFG support for free.

7.2.4 Third Implementation (2022): Uniformity Analysis for Irreducible Control Flow

D130746, *RFC: Uniformity Analysis for Irreducible Control Flow* [31], posted in July 2022 by Sameer Sahasrabuddhe (with the underlying concepts credited to Nicolai Hähnle, and further contributions from Ruiling Song and Jay Foad – an AMD GPU-compiler-team effort throughout), is the analysis that finally closes the gap left open in Section 7.2.2. It is explicitly framed as an extension of Rosemann, Moll, and Hack's abstract interpretation for reducible CFGs [27] (Section 6.5) to irreducible control flow, using the cycle abstraction of Section 7.2.3 in place of natural loops.

Convergence, Not Just Divergence

The central conceptual move in D130746 – and in the LLVM documentation that now formalizes it, *Convergence and Uniformity* [22] – is to stop treating "uniform" as a property of a *value* in isolation, and instead define it in terms of a more primitive relation between **dynamic instances** of an instruction across threads. The rest of this subsubsection builds that relation up from first principles, because the payoff – **maximal convergence** and the m-converged criterion – only makes sense once the underlying mathematics is on the table.

Definition: Static and Dynamic Instances

A **static instance** is a single occurrence of an instruction in the program text. A **dynamic instance** is one particular execution of a static instance by one particular thread. The same static instance (e.g. a `phi` node inside a loop) has many dynamic instances – one per thread, per loop iteration that thread executes. Write $\mathrm{Dyn}(X)$ for the set of *all* dynamic instances of a static instance X, across every thread and every iteration any thread executes it. Everything below is a **relation** R on this set – in the ordinary mathematical sense: a relation R on a set S is simply a specification of which ordered pairs of elements of S count as **related**, and nothing more.

Why does casting the problem this way – a relation R on $\mathrm{Dyn}(X)$, rather than a direct comparison of values – actually buy anything? Look closely at what "do two threads agree on X" is really asking, and it turns out to be two questions wearing one coat. Question (1) is *eligibility*: before any comparing can happen at all, which two dynamic instances of X are even eligible to be compared, i.e. related by R over $\mathrm{Dyn}(X)$? That question has to be settled first, because it decides who is even allowed to sit at the table. Only once it is answered does question (2), *agreement*, get to be asked at all: of the two dynamic instances that made it to the table, did they actually compute the same value for X?

For ordinary structured, acyclic code the answer feels too obvious to state – of course you compare the one and only visit each thread makes to X, because each thread visits X just once.

But a `phi` node inside a loop already has many dynamic instances per thread, one per iteration, and Section 7.2.3 established that once the CFG is irreducible, a cycle can admit more than one

valid header, so even the notion of "which iteration" can stop being well defined relative to a single, canonical hierarchy.

Once eligibility itself is in question, a relation is exactly the right tool, because it keeps the two questions apart. R answers only question (1): which ordered pairs of $\mathrm{Dyn}(X)$ count as related. That question is purely structural – it is computable from the CFG and its cycle forest alone, with no reference to values or execution at all. In particular, R does not depend on which valid cycle hierarchy a particular traversal happened to pick, and it is settled before question (2), value agreement, is even attempted.

Two properties of R turn out to matter here more than any others. A relation R on S is **symmetric** if, whenever x relates to y, y also relates to x – the relationship has no built-in direction. A relation is **transitive** if, whenever x relates to y and y relates to z, x also relates to z – the relationship chains.

Call R itself **executed together**, since that is exactly the intuition it is meant to capture: two dynamic instances $X_1, X_2 \in \mathrm{Dyn}(X)$, produced by two different threads, are executed together precisely when the control-flow connecting them means their threads reached X as part of the same act of reconvergence – the same structural "moment" in the program's execution, even though the two threads may have arrived at X along entirely different paths through the CFG. This is deliberately a structural notion, not a temporal one: it says nothing about wall-clock timing, only about which pairs of visits the control-flow itself ties together.

Once the relation is read this way, symmetric and transitive both follow directly from the definition, with no separate argument needed for either. Symmetry: saying X_1 is executed together with X_2 *is* saying X_2 is executed together with X_1 – both are just two ways of naming the one same act of reconvergence, and that act does not point from one thread to the other. Transitivity is the same observation applied twice: if X_1 and X_2 name the same act, and X_2 and X_3 name that same act, then all three are simply naming it, so X_1 and X_3 are executed together too. Reflexivity holds for an even more trivial reason – a thread's own visit is, of course, part of the same act of reconvergence as itself. Symmetric, transitive, and (trivially) reflexive: "executed together" is therefore an **equivalence relation** on $\mathrm{Dyn}(X)$.

Definition: Converged-With

Converged-with is a relation on $\mathrm{Dyn}(X)$, the set of dynamic instances of a single static instruction X. Two dynamic instances $X_1, X_2 \in \mathrm{Dyn}(X)$, produced by two different threads, are **converged-with** each other – written $X_1 \bowtie X_2$ – when the control-flow connecting them means their threads reached X as part of the same act of reconvergence. **Converged-with** is symmetric, transitive, and reflexive by the argument just given, which makes $\bowtie$ what is called an **equivalence relation** on $\mathrm{Dyn}(X)$: an equivalence relation's defining feature is that it partitions its underlying set into disjoint groups – **equivalence classes** – inside which every pair is related, and outside of which no pair is. Concretely, $\bowtie$ partitions $\mathrm{Dyn}(X)$ into disjoint **convergence classes**, and it is these classes, not individual dynamic instances, that a static analysis can hope to reason about: the question "did these two threads execute X together" always has a well-defined yes/no answer once the classes are fixed, precisely because symmetry and transitivity rule out the incoherent cases above.

Two worked examples fix the intuition.

Example 1: If/Else Reconvergence.

Fix the static instruction X to be the first instruction at the IPDOM (immediate post-dominator) of an ordinary `if/else` branch. Two threads take opposite arms of the branch, and (this being

structured, acyclic code) each executes X exactly once – giving two dynamic instances $X_1, X_2 \in \mathrm{Dyn}(X)$, one per thread. The control-flow connecting X_1 and X_2 means both threads reached X as part of the same act of reconvergence, namely the join point of the branch, so the relation *holds*: $X_1 \bowtie X_2$. Consequently $\{X_1, X_2\}$ is a single convergence class.

Example 2: Loop Iterations.

Fix the static instruction X to be a single instruction inside a loop body, and consider two threads executing that loop.

Case (a) – same iteration. Suppose both threads reach X on the same iteration, with dynamic instances $X_1, X_2 \in \mathrm{Dyn}(X)$. Exactly as in Example 1, the control-flow connecting X_1 and X_2 means both threads reached X as part of the same act of reconvergence – entering that iteration together – so the relation *holds* here too: $X_1 \bowtie X_2$. One convergence class exists per iteration, containing every thread currently executing it.

Case (b) – different iterations. Suppose instead one thread is on iteration i and the other on a different iteration j, with dynamic instances $X_1', X_2' \in \mathrm{Dyn}(X)$ – still dynamic instances of this same static X, since both threads are executing it, just on different passes through the loop. Here the relation *fails*: $X_1' \not\bowtie X_2'$. One thread has passed the loop header strictly more times than the other, so X_1' and X_2' belong to different acts of reconvergence entirely and sit in different convergence classes; comparing their values would be comparing apples to oranges, regardless of how similar the two instructions look on the page.

Converged-with, though, only ever speaks about *one* static instruction at a time: it partitions $\mathrm{Dyn}(X)$ for a single X, and says nothing about how a dynamic instance of X relates to a dynamic instance of some *other* static instruction Y. That is a real gap, because much of what a compiler actually needs to know is cross-instruction: whether a value computed at Y is guaranteed to be available, in whatever sense the analysis cares about, by the time execution reaches X. Closing that gap takes a second relation – one that orders dynamic instances across the *whole* execution, of any static instructions, not just within one instruction's convergence classes – something a later analysis pass can use to reason about "did Y's value get fixed before X needed it," where X and Y may be entirely different static instructions.

Before pinning down what kind of relation this should be, fix the set it lives on, since that is exactly where converged-with does *not* help: converged-with is a relation on $\mathrm{Dyn}(X)$, the dynamic instances of one fixed X, and cross-instruction ordering has no use for that restriction. Call $\mathcal{U}$ the **universe** – the set of *every* dynamic instance produced anywhere in the execution, of any static instruction whatsoever. Concretely, $\mathcal{U}$ is the union of $\mathrm{Dyn}(X)$ over every static instruction X in the program, not just one. The elements P and Q that appear below are arbitrary elements of this universe $\mathcal{U}$: each is some thread's one execution of some static instruction, but unlike $X_1, X_2 \in \mathrm{Dyn}(X)$ for converged-with, P and Q need not be instances of the *same* static instruction at all.

Which ordering relation holds on the universal set $\mathcal{U}$? Three candidates are worth checking, in order.

Does a Non-Strict Order Hold on $\mathcal{U}$?

A non-strict order is a relation "$\leq$" that is reflexive, antisymmetric, and transitive. Reflexivity costs nothing: declaring $P \leq P$ for every $P \in \mathcal{U}$ rules out nothing. Antisymmetry is where it fails: take clock time as "$\leq$". Converged threads in the same SIMT group execute the *same* instruction on the *same* cycle, using a single shared program counter (Section 1), so two *distinct* dynamic instances

P, Q share a timestamp – giving $P \leq Q$ and $Q \leq P$ without $P = Q$, a direct antisymmetry violation. So: no non-strict order can be built on $\mathcal{U}$.

Does a (Strict) Total Order Hold on $\mathcal{U}$?

Only the strict dressing remains. A **strict total order** is a relation "$<$" that is irreflexive and transitive, and additionally satisfies **trichotomy**: for every two *distinct* elements $x, y \in S$, exactly one of $x < y$ or $y < x$ holds – a different axiom from antisymmetry, so it needs its own check. A single thread's own program order satisfies it: one instruction at a time, so any two of its dynamic instances are always comparable.

That breaks the moment more than one thread is involved. Converged threads in the same SIMT group execute the *same* instruction on the *same* cycle, on a single shared program counter (Section 1): their dynamic instances are *literally simultaneous*, so neither precedes the other, and trichotomy demands that exactly one of $P < Q$ or $Q < P$ hold. One such pair is already enough to break a claim that has to hold for *every* pair: no strict total order can be defined on $\mathcal{U}$.

Does a Strict Partial Order Hold on $\mathcal{U}$?

Three of the four combinations of {strict, non-strict} × {total, partial} are now eliminated: non-strict order failed outright, in both its total and partial forms, on antisymmetry; strict order failed in its total form, on trichotomy. Exactly one combination remains untested – strict *and* partial together – which requires proving exactly two things: irreflexivity, and transitivity.

Irreflexivity: an ordering relation should never say an element precedes itself. Nothing seen so far touched this – a dynamic instance was never in question against itself in the first place – so there is no reason to give it up, and every reason to keep it: an order that let something precede itself would not be describing "before" at all.

Transitivity: if P is before Q, and Q is before R, then P must be before R. This, too, survives every objection raised so far untouched.

So: yes. A strict partial order on $\mathcal{U}$ is exactly what survives once totality is abandoned and the relation is built from the right kind of fact – structural, not physical. That relation is **convergence-before**.

Definition: Convergence-Before

Convergence-before is a relation on the universal set $\mathcal{U}$ – the set of all dynamic instances of every static instruction in the program, not just one. It is built out of converged-with by three rules, applied repeatedly until nothing new follows.

(1) Same thread. If two dynamic instances $P, Q \in \mathcal{U}$ are both from the same thread, and that thread executes P before Q, then $P \prec_c Q$.

(2) Crossing threads, forward. Say two dynamic instances $Q_1, Q_2 \in \mathcal{U}$ satisfy $Q_1 \bowtie Q_2$, with Q_1 from thread 1 and Q_2 from thread 2. If $P \in \mathcal{U}$ is a dynamic instance also from thread 1 with $P \prec_c Q_1$, then $P \prec_c Q_2$ too: P's precedence over Q_1 carries across to Q_2, because Q_1 and Q_2 are the same act of reconvergence.

(3) Crossing threads, backward. Say two dynamic instances $P_1, P_2 \in \mathcal{U}$ satisfy $P_1 \bowtie P_2$, with P_1 from thread 1 and P_2 from thread 2. If $Q \in \mathcal{U}$ is a dynamic instance also from thread 2 with $P_2 \prec_c Q$, then $P_1 \prec_c Q$ too.

Like converged-with, none of these three rules inspects a runtime trace – all are static, structural facts about the CFG, which is exactly what makes convergence-before something a compiler can compute ahead of time.

Two relations are now built: converged-with on $\mathrm{Dyn}(X)$, and convergence-before on $\mathcal{U}$, with convergence-before built entirely *out of* converged-with via rules (1)–(3). This order matters: convergence-before is only as trustworthy as the converged-with it is built from, so converged-with has to be pinned down correctly on each $\mathrm{Dyn}(X)$ *before* convergence-before on $\mathcal{U}$ can be trusted at all. What is needed next is a check of exactly what goes wrong when it is not: convergence-before earned the name strict partial order only because it is irreflexive, meaning no dynamic instance may ever precede itself. Transitivity is guaranteed structurally, simply by how rules (1)–(3) chain – but irreflexivity is not automatic in the same way, since it depends entirely on which pairs $\bowtie$ declares "together" in the first place.

Here is concretely how a careless $\bowtie$ breaks it. Let X_1 and X_2 be two dynamic instances of the same static instruction X, with X_1 from thread 1 and X_2 from thread 2 – $\bowtie$ only ever relates instances from two different threads to begin with. Suppose $\bowtie$ declared $X_1 \bowtie X_2$ more generously than the control-flow actually supports – that is, without a real act of reconvergence tying the two threads' visits to X together.

Now bring in two facts that have nothing to do with this bad pair. First, thread 2's own program order already gives $X_2 \prec_c Y$ for some dynamic instance Y that thread 2 executes after X_2 – rule (1), true regardless of what $\bowtie$ says. Second, suppose some entirely separate, genuinely correct act of reconvergence elsewhere in the program already established $Y \prec_c X_1$: a legitimate cross-thread convergence-before fact, reaching from thread 2 back to thread 1.

Rule (3) now takes the bad pair $X_1 \bowtie X_2$ together with $X_2 \prec_c Y$ and concludes $X_1 \prec_c Y$. But $Y \prec_c X_1$ was already established, correctly. Chaining the two together – $X_1 \prec_c Y$ and $Y \prec_c X_1$ – forces $X_1 \prec_c X_1$: some dynamic instance ending up convergence-before itself. A strict partial order forbids this outright, by definition, since nothing may precede itself.

Convergence-before on $\mathcal{U}$ is a strict partial order, which means acyclicity must hold for every pair it relates. The example just given shows exactly how a carelessly chosen $\bowtie$ on $\mathrm{Dyn}(X)$ can break that: declaring $X_1 \bowtie X_2$ without a real act of reconvergence created a cycle, forcing $X_1 \prec_c X_1$. So converged-with cannot be established carelessly – whatever $\bowtie$ is chosen on each $\mathrm{Dyn}(X)$, it must not break the acyclicity that convergence-before on $\mathcal{U}$ requires.

The next step, then, is to find a way of choosing $\bowtie$ on each $\mathrm{Dyn}(X)$, for every static instruction X in the program, that is guaranteed by construction to keep the convergence-before built from it acyclic – not a relation that has to be checked for safety after the fact, but one that is correct by how it is defined. That way of choosing $\bowtie$ is exactly what **maximal convergence** names, and defining it precisely is the subject of the rest of this subsubsection.

Maximal Convergence and the m-Converged Criterion

Maximal convergence is the mechanism that settles this: one uniform recipe for constructing $\bowtie$ on $\mathrm{Dyn}(X)$, for every static instruction X in the program, such that the very act of building $\bowtie$ this way keeps the convergence-before built from it acyclic. The recipe is defined recursively, over the cycle forest of Section 7.2.3. Take two dynamic instances $X_1, X_2 \in \mathrm{Dyn}(X)$, produced by different threads: this recipe declares $X_1 \bowtie X_2$ if and only if X lies in no cycle at all (the base case); or, for *every* cycle C with header H that contains X, every dynamic instance of H that precedes X_1 in X_1's own thread is convergence-before X_2, *and* every dynamic instance of H that precedes X_2 in X_2's own thread is convergence-before X_1. In prose: outside any cycle, two threads reaching the same instruction are simply granted convergence – this base case is all that ordinary structured code (if/else diamonds and the like) ever needs. Inside a cycle, the definition demands a mutual "neither thread is strictly ahead of the other, relative to this cycle's header" check – and it states that check using convergence-before itself, rather than by literally counting loop iterations.

Maximal convergence's definition, inside a cycle, is stated in terms of that cycle's header H.

But a header is not always unique: for an irreducible cycle, more than one entry can validly serve as the header (Section 7.2.3). So the same pair X_1, X_2 could, in principle, get different converged-with verdicts depending on which valid header is chosen. That choice is an implementation detail – which header the underlying cycle-detection algorithm happened to pick – not a fact about the program's control flow or the hardware's behavior. A compiler cannot let whether two threads are converged depend on an implementation detail like that.

m-converged names the fix: it is the subset of static instructions whose maximal-convergence classification comes out the same no matter which valid header is chosen. For these instructions, the classification can be trusted outright. Two structural criteria identify m-converged instructions in practice:

- **Diverged-entry criterion.** For a divergent branch B with join node J, nodes of a cycle C containing both B and J are m-converged only if at least one of the following holds:
  - B strictly dominates J;
  - the header of C strictly dominates J;
  - this same condition holds recursively for some cycle nested inside C.
- **Diverged-paths criterion.** A cycle's entries are the nodes reachable from outside it without passing through any other node of the cycle first; an irreducible cycle has more than one entry, and its header is only one of them – whichever a depth-first search happens to visit first (Section 7.2.3). Suppose a divergent branch sends different threads down different paths, and those paths reach a cycle C by entering it at two different entries. Then which of those entries counts as C's header is exactly the arbitrary hierarchy choice described above, and there is no way to say, independently of that choice, whether the two groups of threads are synchronized. So C's contents are conservatively classified as not m-converged.

Reducible CFGs satisfy m-convergence everywhere, since a reducible cycle has only one entry and therefore only one possible header: the independence condition holds trivially. Every other static instruction – one whose maximal-convergence classification could depend on which header was chosen – is conservatively classified as not converged, and therefore as divergent (Section 6).

Definition: Uniformity, Restated

The output of a static instruction X is **uniform** for a set of threads if and only if X_1 and X_2 produce the same value whenever $X_1 \bowtie X_2$, for every pair of dynamic instances of X those threads produce. Restated in terms of convergence classes: $\mathrm{Dyn}(X)$, restricted to these threads, splits into disjoint convergence classes, and X is uniform exactly when every member of each class produces the same value – for every one of these classes, possibly a different value for each class.

For a reducible CFG, every static instruction trivially satisfies the m-converged criterion: a reducible cycle has only one entry, so there is only one possible header, and no ambiguity to check. Maximal convergence therefore computes $\bowtie$ exactly, and uniformity is fully determinable for every X.

For an irreducible CFG, this is checked instruction by instruction, cycle by cycle, against the two structural criteria above. An instruction that satisfies them is m-converged and is treated exactly as in the reducible case: maximal convergence computes $\bowtie$ exactly, and its uniformity is determinable. An instruction that fails them is conservatively classified as not converged, and therefore as divergent (Section 6).

Implementation

D130746 implements exactly the mathematics built up in this subsection: converged-with, convergence-before, maximal convergence, and the m-converged criterion, which together determine, for each static instruction, whether its classification can be trusted outright or must fall back to the conservative divergent default. This computation is packaged as an **instantiable template**, not a one-off pass, so the identical algorithm serves both LLVM IR and Machine IR from a single codebase – an explicit design improvement over the 2015 and 2018 implementations, both of which existed only at the LLVM IR level, requiring backend-specific ad hoc treatment of divergence post-instruction-selection.

Concretely, the current (post-D130746) source layout is:

- `llvm/include/llvm/ADT/GenericUniformityInfo.h` – `GenericUniformityInfo<ContextT>`, the generic template exposing the public query API (`isDivergentAtDef`, `isUniformAtDef`, `isDivergentAtUse`, `hasDivergentTerminator`, `getTemporalDivergenceList`, and cache-invalidation via `forgetValue`);
- `llvm/include/llvm/Analysis/UniformityAnalysis.h` and `llvm/lib/Analysis/UniformityAnalysis.cpp` – the LLVM-IR instantiation, exposed to the new pass manager as `UniformityInfoAnalysis`;
- `llvm/lib/CodeGen/MachineUniformityAnalysis.cpp` – the parallel Machine-IR instantiation, used by backends (chiefly AMDGPU) that need divergence information after instruction selection; and
- `GenericUniformityAnalysisImpl<ContextT>` – the shared implementation class computing the fixpoint itself, parameterized over the IR-specific `GenericSSAContext` from Section 7.2.3.

The patch was accompanied by a public LLVM Developers' Meeting technical talk [30] and landed in LLVM trunk in December 2022 with broad support from the AMDGPU backend community, which had the most immediate practical stake in correct irreducible-CFG handling.

7.2.5 Retiring the Legacy Analyses (2023)

With the new Uniformity Analysis passing the full legacy divergence-analysis test suite and covering strictly more cases, the two earlier implementations were no longer needed. **D148116**, *[Analysis] Remove DA & LegacyDA* [26], authored by Pierre van Houtryve in April 2023 following an RFC discussion on LLVM Discourse, removed `DivergenceAnalysis.{h,cpp}`, `LegacyDivergenceAnalysis.{h,cpp}`, and `SyncDependenceAnalysis.{h,cpp}`, migrating their test coverage into the Uniformity Analysis test directories. The patch's stated rationale: *"UniformityAnalysis offers all of the same features and much more, there is no reason left to use the legacy DAs."* What remained afterward is the Uniformity Analysis of Section 7.2.4 – built on the converged-with, convergence-before, and maximal-convergence concepts developed there – which is the sole divergence/uniformity analysis in LLVM trunk as of this writing.

8 Conclusion

Branch divergence is what happens when a programming model built on the fiction of independent threads meets hardware built on the reality of a shared program counter. This paper surveyed

Implementation	Algorithm	Irreducible CFGs	Scope
D8576 (2015)	Graph reachability over data + sync dependence	Not handled soundly	LLVM IR only
D50433 (2018)	Generic worklist DA + lazy SDA	Conservative only	LLVM IR only
D130746 (2022)	Abstract interpretation via maximal / m-convergence over cycles	Sound, precise	LLVM IR *and* Machine IR (templated)

Table 5: The three LLVM divergence/uniformity analysis implementations compared.

the problem and its handling as a single, continuous narrative: starting from the classical control-flow-graph theory of Allen, Hecht, Ullman, and Tarjan – reducibility and structured control flow, worked out for entirely sequential reasons in the 1970s – and connecting it to whether SIMT hardware's immediate-post-dominator reconvergence mechanism behaves well, badly, or (in the worst case) incorrectly. From there, the survey covered the two compiler-side responses built against it: reshaping the control-flow graph via linearization/structurization, and statically classifying values and branches as uniform or divergent via divergence analysis.

Section 7 then traced how divergence/uniformity analysis was implemented inside LLVM specifically, across three successive implementations over eight years – the original 2015 graph-reachability analysis; Simon Moll's 2018 generic sync-dependence framework, restricted to reducible graphs; and the current (2022) cycle-based Uniformity Analysis, which handles irreducible control flow soundly via a purpose-built cycle abstraction. Each implementation was motivated by a specific, namable limitation of its predecessor, each is traceable to a specific, citable source – an academic paper for the first, and public LLVM RFCs for the second and third, the latter an AMD GPU-compiler-team effort – and each was documented, debated, and eventually superseded in the open, on a public issue tracker.

References

[1] Mojtaba Abaie Shoushtary, Jordi Tubella Murgadas, and Antonio Gonzalez. Control flow management in modern GPUs. *arXiv preprint arXiv:2407.02944*, 2024.

[2] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[3] John R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 177–189, 1983.

[4] Jayvant Anantpur and R. Govindarajan. Taming control divergence in GPUs through control flow linearization. In *Proceedings of the 23rd International Conference on Compiler Construction (CC)*, pages 133–153, 2014.

[5] Larry Carter, Jeanne Ferrante, and Clark Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–114, 2003.

[6] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintão Pereira, and Wagner Meira. Divergence analysis and optimizations. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 320–329, 2011.

[7] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 25–36, 2011.

[8] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 407–420, 2007.

[9] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):1–37, 2009.

[10] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, 2010.

[11] Nicolai Hähnle. Analysis: Add a GenericCycleInfo analysis. LLVM Phabricator Differential Revision D83094, 2020. https://reviews.llvm.org/D83094. Based on Havlak's cycle-forest algorithm [12].

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

[13] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal on Computing*, 1(2):188–202, 1972.

[14] Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, 1974.

[15] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 19, pages 1031–1052, 1997.

[16] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Notices*, 29(6):171–185, 1994.

[17] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 141–150, 2011.

[18] Ralf Karrenberg and Sebastian Hack. Improving performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, pages 1–20, 2012.

[19] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 3rd edition, 2017.

[20] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1): 121–141, 1979.

[21] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[22] LLVM Project. Convergence and uniformity. LLVM Documentation, 2024. https://llvm.org/docs/ConvergenceAndUniformity.html.

[23] Simon Moll. A new divergence analysis for LLVM. LLVM Phabricator Differential Revision D50433; RFC on llvm-dev, 2018. https://reviews.llvm.org/D50433.

[24] Simon Moll and Sebastian Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 543–556, 2018.

[25] Carl D. Offner. Notes on graph algorithms used in optimizing compilers. Technical report, University of Massachusetts Boston, 2013.

[26] Pierre van Houtryve (Pierre-vh). [analysis] remove DA & LegacyDA. LLVM Phabricator Differential Revision D148116, 2023. https://reviews.llvm.org/D148116. Committed as rGae77aceba5ad.

[27] Julian Rosemann, Simon Moll, and Sebastian Hack. An abstract interpretation for SPMD divergence on reducible control flow graphs. *Proceedings of the ACM on Programming Languages (POPL)*, 5:1–36, 2021.

[28] Sameer Sahasrabuddhe. Introduce fix-irreducible pass. LLVM Phabricator Differential Revision D77198, 2020. https://reviews.llvm.org/D77198.

[29] Sameer Sahasrabuddhe and Nicolai Hähnle. CycleInfo: Introduce cycles as a generalization of loops. LLVM Phabricator Differential Revision D112696, 2021. https://reviews.llvm.org/D112696.

[30] Sameer Sahasrabuddhe and Nicolai Hähnle. Uniformity analysis for irreducible CFGs. 2022 LLVM Developers' Meeting, Tech Talk, 2022.

[31] Sameer Sahasrabuddhe, Nicolai Hähnle, Ruiling Song, and Jay Foad. RFC: uniformity analysis for irreducible control flow. LLVM Phabricator Differential Revision D130746, 2022. https://reviews.llvm.org/D130746.

[32] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 35(4):1–36, 2014.

[33] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. DARM: Control-flow melding for SIMT thread divergence reduction. In *Proceedings of the 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 186–196, 2022.

[34] Robert E. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 96–107, 1973.

[35] Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. In *International Journal of High Performance Computing Applications (IJHPCA)*, volume 26, pages 170–185, 2012.

[36] Jingyue Wu. Divergence analysis for GPU programs. LLVM Phabricator Differential Revision D8576, 2015. https://reviews.llvm.org/D8576. Committed as r234567.

[37] Feng Zhang and Erik H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, 2004.