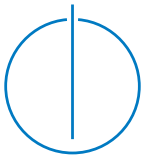
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Vectorising Divergent Control-Flow for SIMD Applications

Fabian Wahlster



DEPARTMENT OF INFORMATICS

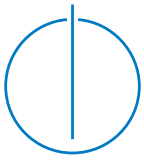
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Vectorising Divergent Control-Flow for SIMD Applications

Vektorisieren von Divergentem Kontrollfluss für SIMD-Anwendungen

Author:	Fabian Wahlster
Supervisor:	Prof. Michael Gerndt
Advisor:	Dr. Nicolai Hähnle
Submission Date:	14.12.2018



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, 14.12.2018 Fabian Wahlster

Acknowledgments

I would like to thank Nicolai Hähnle and AMD for the continued interest and support for this work. His insights allowed me to dive deeper into the inner workings of contemporary GPU architectures and compiler design, especially divergence management for the AMDGPU backend of LLVM. I would also like to thank Moritz Becher and Erik Jansonn for the insightful discussions and feedback on the topic as well as Alex Vlachos from Valve for providing a perfect example shader.

Abstract

Vectorization of divergent control flow is a common problem of code generation for SIMD applications. The widespread programming model of high-level GPGPU languages is based on a *thread-level* view of the program, where it looks like it is operating on scalar data (SIMT, SPMD programming model). In reality, the underlying hardware executes many program instances in parallel bundled in thread groups (wavefront in AMD's terms or warp for Nvidia) that execute the same instruction on multiple data in a lock-step manner. Because all threads of such a group share the same program counter, conditional branching for individual instances is not possible. A common approach to this problem is executing all code paths, but selecting the result in each program instance separately based on the branch condition. This behavior can be implemented by masking the execution of vector instructions to remain dormant for inactive instances not satisfying the branch condition. Executing both branches of an if-statement reduces the throughput and utilization of the hardware. Only when the branch condition evaluates to the same result for all instances, uniform control flow is possible, allowing optimal utilization.

Different hardware and software based solutions have been implemented to effectively vectorize control flow for SIMD applications. Software approaches have been proposed that linearize the control flow in the compiler. Several hardware solutions implement divergence management using reconvergence stacks to keep track of the active threads in a group, hiding most of the complexity from the compiler backend and allowing unstructured code. Both approaches rely on a form of execution masking available in hardware.

This work describes a software-based masking approach, and implements a two-pass algorithm to lower *thread-level* control flow to *wave-level* control flow by first transforming the input CFG such that rendezvous points for disjoint threads exist, and then uses this property of a reconverging CFG to generate the appropriate execution mask instructions for vectorization of divergent branches. The notion of a *reconverging* CFG is a simple, yet powerful tool to structurize control flow for GPUs and other SIMD applications. Some of the key features of the algorithm are: Allowing arbitrary irreducible input CFGs and preservation of uniform branches without duplicating code while leaving the input unmodified if it is already reconverging.

Contents

1. Introduction

1.1. Hardware

For more than a decade (with the rise of consumer graphics cards in the 1990s until the specification of Direct3D 9 in 2002), the sole purpose of Graphics Processing Units (GPUs) was to render computer graphics by interpolating and rasterizing triangles with fixed function hardware (Nvidia NV1, 3Dfx Voodoo or ATI Rage 128). Those chips where specifically designed for graphics applications and offered hardwired texture and shading functions (e.g. Gouraud, T&L transform and lighting). Since then more and more parts of the graphics pipeline, starting with the vertex and pixel stage, became programmable (with DirectX v8 on IR assembly level and HLSL on version 9.0) and later merged into a unified shader model with DirectX 10, paving the way for General Purpose GPUs (GPGPUs), a term coined by Nvidia's hardware architect Mark Harris. GPUs are now broadly used for non-graphics applications such as high-performance computation for scientific simulations and machine learning with different programming models and APIs available (OpenCL, CUDA), surpassing CPUs with their immense parallel computing capabilities.

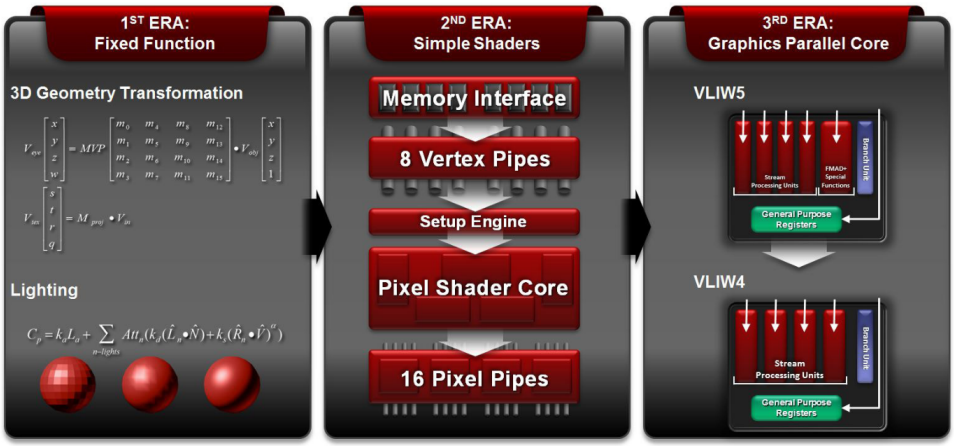


Figure 1.1.: Different historical eras of ATI/AMD GPUs up until GCN architecture (Image source [AMD]).

GPUs are ultra-wide SIMD machines, but from a developers point of view, their programs are executed as if they were operating on scalar data (later referred to as 'thread-level' view), though the program instances actually run in parallel on the hardware (referred to as 'wave-level' for AMD GPUs). Nvidia calls this technique SIMT: Single Instruction Multiple Threads whereas Intel refers to this as SPMD (Single Program Multiple Data), but the programming model is the same for most parallel accelerator hardware. To achieve this amount of parallelism in hardware, Functional Units (FUs) are replicated in high volume, but their complexity does not hold up to conventional multi-purpose CPUs with deep pipelines, branch prediction and out-of-order execution.

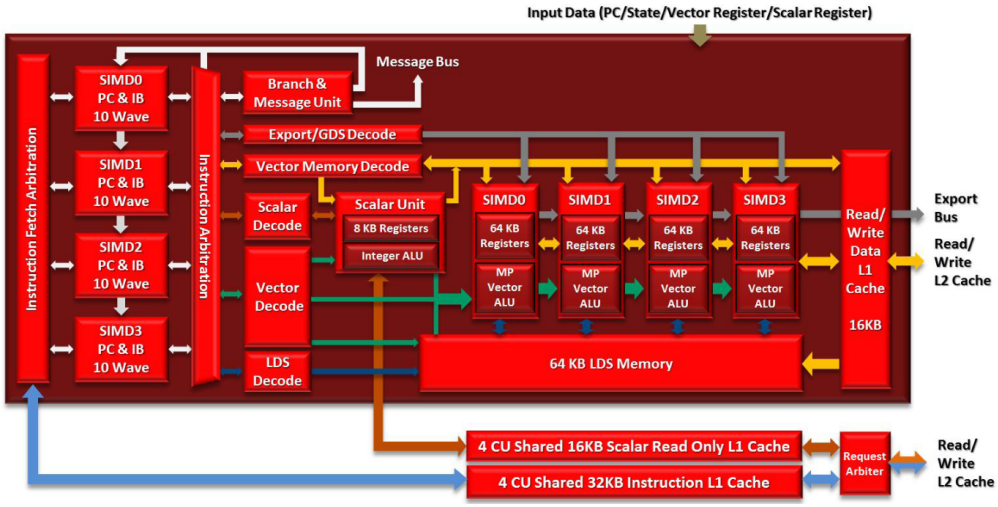


Figure 1.2.: First generation GCN Compute Unit layout (Image source [AMD]).

Looking at the GCN layout overview of Figure 1.2 we can see how this is realized for AMD GPUs and understand the architectural differences to CPUs and their vector units: The GCN processor is divided into several compute units (CUs), each with a number of vector ALUs (4 for the Vega generation) and a single scalar ALU with their dedicated registers. Up to 10 waves can be scheduled to use one vALU through separate Program Counters (PC). In a way, waves are scheduled within a CU just like threads are scheduled in a CPU core with Simultaneous Multi-Threading (SMT). A wavefront contains all work-items scheduled for this unit which execute the same instruction in a lock-step manner. As a result, control flow which diverges at program branches is handled on an instruction level using predication: scalar instructions (working on scalar registers) operate on one value per wavefront and handle all control flow while vector instructions operate on all work-items in the wavefront at once, with unique values for

each item. An execution mask (64b EXEC mask register, each bit represents a vector lane) is used to determine which work-items are active in the current instruction (as seen in Figure 1.3) and which are dormant and therefore treated as NOP (no operation). Scalar ALU instructions can be used to manipulate EXEC masks and thereby implement vector control flow.

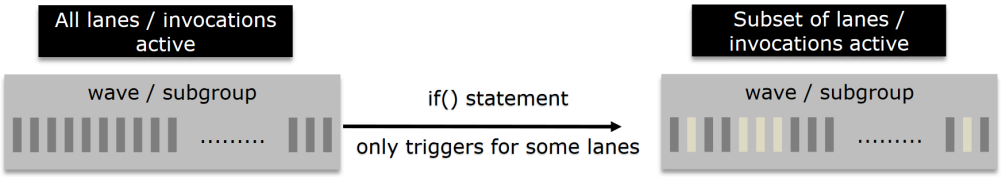


Figure 1.3.: Effect of branch divergence on wave lane invocations (Image source [LG]).

1.2. Compilation

This work will focus on the main aspects of control flow transformation for highly parallel processors. One of the compiler's main tasks for supporting the described programming model is to parallelize serial programs by transforming thread-level to wave-level control flow. For current AMD hardware that also implies manipulating the execution mask and inserting scalar branching instructions where needed, vectorising the program for this ultra-wide SIMD machine. Manual vectorization (of loops) and vectorising compilers have been used for HPC applications for quite some time. Intel's ISPC Project [PM12] brings programming concepts known from GPU and shader development to the SIMD extension of the CPU and generate vector instructions [Ken] for data-parallel workloads. Unity's 'math-aware' LLVM-based Burst compiler backend [Kro] is another relevant example of technology emerging in this area. Apart from the required transformations to reflect the programming model, the whole compilation process for GPUs differs greatly from the common approach for CPUs.

```
float fn0(float a,float b)
{
   if(a>b)
      return((a-b)*a);
   else
      return((b-a)*b);
}
```

(a) Thread-level developers point of view from a shading language perspective

```
//Registers r0 contains "a", r1 contains "b"
//Value is returned in r2

   v_cmp_gt_f32      r0,r1          //a > b, establish VCC
   s_mov_b64         s0,exec        //Save current exec mask
   s_and_b64         exec,vcc,exec  //Do "if"
   s_cbranch_vccz    label0         //Branch if all lanes fail
   v_sub_f32         r2,r0,r1       //result = a - b
   v_mul_f32         r2,r2,r0       //result=result * a

label0:
   s_andn2_b64       exec,s0,exec   //Do "else"(s0 & !exec)
   s_cbranch_execz   label1         //Branch if all lanes fail
   v_sub_f32         r2,r1,r0       //result = b - a
   v_mul_f32         r2,r2,r1       //result = result * b
label1:
   s_mov_b64         exec,s0        //Restore exec mask
```

(b) GCN ISA wave-level representation

Figure 1.4.: Transforming high-level shader code to GCN ISA with execution mask instructions (Image source [MH]).

While Ahead-of-Time compilers, targeting common microprocessor architectures like x86 or ARM, directly emit machine byte code that can be executed by the operating system (also because the ISA is publicly available), the final binary for GPU programs is not visible to the application developer because of the Run/Load Time Compilation process of shaders [Chi17]. Shader programs are translated from special purpose languages (HLSL, GLSL, CG) to intermediate representations (DXIL, SPIR-V, LLVM-IR) and in a later step lowered to the actual machine code by the device driver. This process is invoked for example when a graphics API (like Vulkan® or DirectX) wants to bind the program to the graphics pipeline during the application runtime. For compute applications running on OpenCL or CUDA the process is similar, but the IR (in form of PTX or HSA IL) can be packed into the host executable and is finalized by the driver's compiler before execution. Instead of an intermediate representation of the compute kernel, AMD's ROCm platform is able to pack target ISA into the binary.

1.3. Problem description

This thesis explores a new approach for control flow transformations in the context of AMD's continued open-source efforts where hardware architecture and programming models are available under the GPUOpen program, though all proposed concepts are applicable to other SIMD hardware architectures as well. The LLVM framework builds the foundation of AMD's compiler backend (see Figure 1.5) used in the open-source GPU drivers for Linux, therefore algorithms presented in this work will be implemented as part of LLVM.

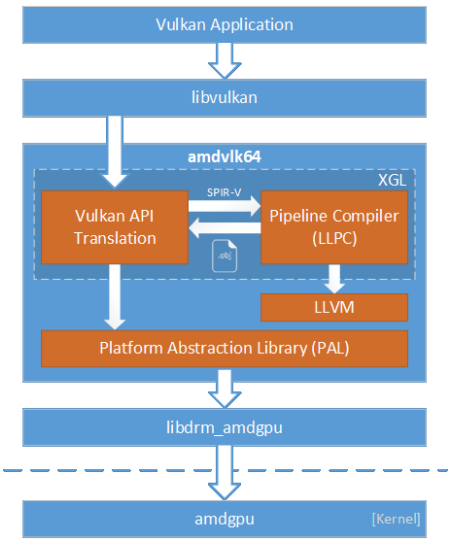
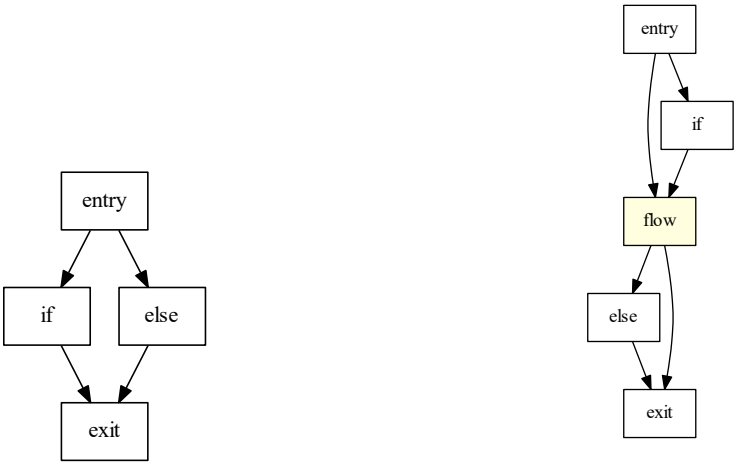


Figure 1.5.: AMDVLK: AMD Open Source Driver for Vulkan. Image source [Hea].

Branch divergence is a common performance problem for GPU programs [Dia+11; Col; Lee+14], ever increasing with nested branches and loops. An important property of control flow is whether it is uniform, meaning that if the evaluation of a branch condition entirely depends on constants or can be derived to be constant over all invocations. Uniform control flow is divergence free, hence more efficient scalar branching instructions can be used to skip entire blocks instead of executing masked vector operations where control flow can not be interleaved to utilize parallelism among the wave.

GPU vendors have to tackle this problem differently because many lanes, corresponding to many threads of the original program, share a single PC. Special hardware stacks [AR14; Col; MH; Dia+11] can be used to allow unstructured code input, but they require additional circuits (and therefore die space) and special programming to manage divergence. Software controlled execution masking methods on the other hand work on generalized hardware that can be reused (e.g. for ALU operations). The question remains whether which approach is better suited to alleviate this problem and whether it is useful to shift complexity between hardware and software. The algorithm proposed here is designed to work with any SIMD hardware that supports masked execution, making it applicable to not just GPUs but for example also AVX512 extension capable processors.



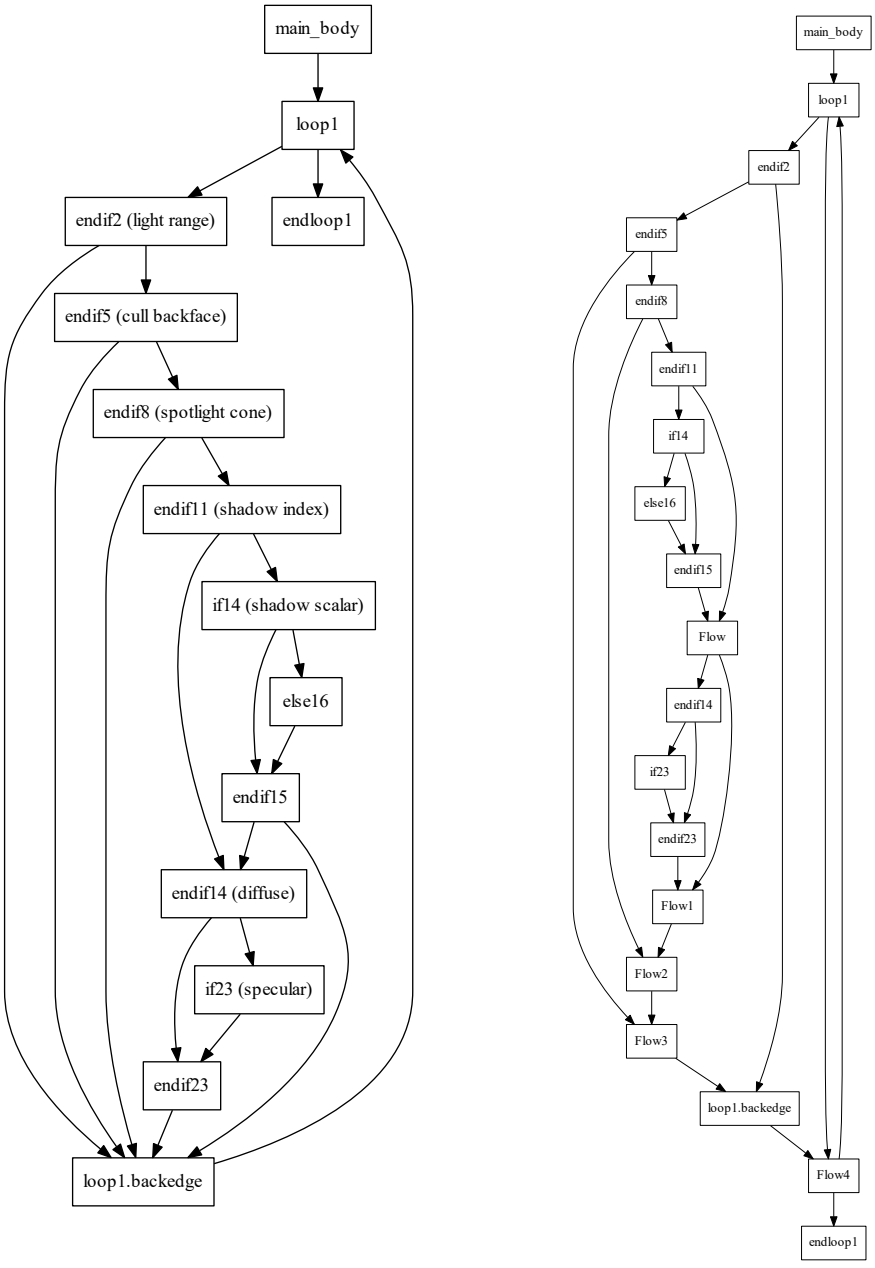
(a) Control flow for a simple if-block

(b) Structured control flow with additional flow-block inserted

Figure 1.6.: Structurization can be used to express wave-level control flow.

Unstructured control flow can originate from code containing early-exit operations like **goto**, **break** and **continue**. Different definitions of *structured control flow* exist [SSE17] (one of which requires *single entry-single exit* blocks), making it hard to formalize its influence on the wave-level flow that is required by the hardware. Structured control flow guarantees to produce reducible control flow graphs (CFG), meaning that loops have exactly one entry, improving the situation where optimizers fail to vectorize irreducible loops with several entries and exit points. The basic CFG example of an **if** block in Figure 1.6a shows that control flow either passes through the **if** or the **else** part, for wave-level control flow however (Figure 1.6b), both blocks must be traversed if threads diverge in the wave. This is not possible in the first example. We see that structuring the original CFG transforms it to reflect wave-level flow, but other (possibly simpler) transformations with the same effect might exist.

This problem is an active area of research (see Chapter 3) and other transformations and control flow linearization techniques are being investigated for handling divergent flow on GPUs. When considering uniform control flow however, structurization can actually hinder performance optimizations due to unneeded branches being introduced when actually all invocations would perform the same operation (or the whole block could be skipped using a s_cbranch instruction as seen in Figure 1.4b). Structurization also influences the lifespan of registers, possibly changing register allocation behaviour



(a) Control flow graph of a shader from Valve (Listing B.1).

(b) Shader structurized using LLVM's StructurizeCFG region pass.

Figure 1.7.: Real-world example shader structurized by the current LLVM implementation.

to be too conservative for uniform control flow.

Figure 1.7 shows a control flow graph of a shader used in a VR application written by Valve. LLVM's *StructurizeCFG* pass currently used to prepare the lowering of control flow operates on Single Entry Single Exit (SESE) basic block regions, and misses the opportunity to analyse the control flow from a whole-function point of view. This leads to unnecessary addition of *flow* blocks and backward edges (Figure 1.7b), even though the shader is already in a good structure for lowering to wave-level control flow.

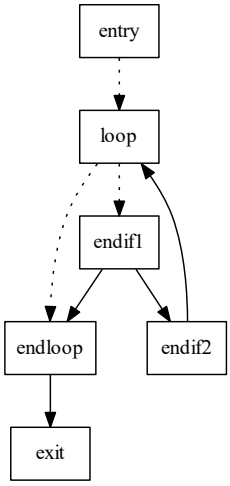
This thesis evaluates if the rather strong transformation of structurization is actually required for wave-level code, or if a weaker notion of *reconvergence* can be formulated such that diverging threads can be re-joined, and uniform control flow can be handled properly.

The main contributions of this work are:

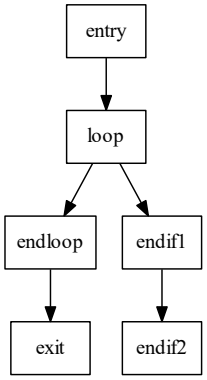
- Concise definition of a reconverging CFG
- Lowering algorithm exploiting the properties of such a CFG
- Algorithm transforming arbitrary and irreducible input CFGs to contain reconvergance points while also preserving uniform control-flow
- Evaluation of basic block ordering methods used as input of the CFG transformation

2. Background

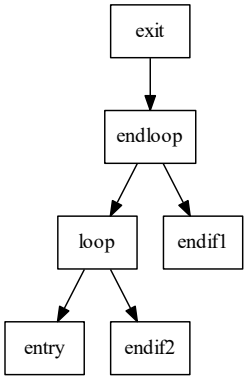
Code can be represented as a graph of control flow based on the branch instructions found in the program. Control flow graphs are important tools for various program optimization techniques. Other representations, such as data flow graphs (used for dependency and divergence analysis) are related and often used in compiler backends. This chapter revisits some basic definitions (following [SSE17]) and building blocks required for transformations of control flow graphs and their analysis.



(a) Example CFG. Dotted edges represent uniform flow.



(b) Dominator tree of CFG 2.1a.



(c) Post-dominator tree of CFG 2.1a.

Definition 1. *A control flow graph is comprised of* ***basic block****s of instructions. control flow only enters a basic block through its first instruction and leaves it at its last instruction (called* terminator*) without any branching in-between. Branch instructions can only occur at the terminator of each basic block, building the edges of the directed flow graph. Each conditional branch terminator has two or more successors.*

Definition 2. *The source of the graph is called **entry** node while a node at which the program terminates is called **exit** node. A control flow **path** is a sequence of adjacent nodes and edges connecting two basic blocks A and B, a region contains all paths that lead from a A to B.*

Definition 3. *Any node whose basic block terminates with a conditional branch with two or more out-going edges is considered a **condition node**. If such a condition node N is the source and sink of a simple **loop path** (each node, except the condition node, has in/out-degree of one), it is called **loop condition node**.*

Basic blocks which cannot be reached from the entry node are referred to as *dead code*. Dead code can be the result of transformations on the CFG that rewire branches or user code with early return statements for instance.

Definition 4. *A **dominator** P of Q is a node where every path from the entry node of the CFG has to pass through to connect to Q (P dominates Q). A node (or edge) P **post-dominates** Q if every path from Q to the exit node of the CFG has to pass through P. These relationships can be used to construct (post-) dominator trees (see Figures 2.1b and 2.1c) where the dominators of each node are exactly the ancestors in the dominator tree. The parent node of a post-dominator tree is called immediate post-dominator (IPDOM).*

Definition 5. *If a node A dominates a node B and B post-dominates A and every cycle containing A also contains B (and vice versa), the region between A and B is called **single-entry-single-exit** (SESE).*

Definition 6. *A CFG is **reducible** if it can be coalesced into one single node after repeated application of the following two transformations:*

T1 Removing edges of a node onto itself.

T2 Merging two nodes A and B into one if A is the only predecessor of B. The merged node retains A's predecessors and B's successors.

Otherwise, its irreducible.

(a) CFG before application of T1 and T2 (b) CFG after application of T1 and T2

Figure 2.2.: Irreducible CFG that can not be coalesced into a single node.

The term *Structurization* for CFGs derives from *Structured programming* [Knu74] [CD78] which is concerned with the high-level language aspects (e.g. whether to allow goto statements) rather than the composition of basic blocks in a flow graph. Various definitions of what it means for CFGs to be structured exist [AR14] [Wil77] and current compilers and languages like LLVM [LLVc] and WebAssembly [Haa+17] implement structurization with their own means (constructing SESE regions vs. scope nesting).

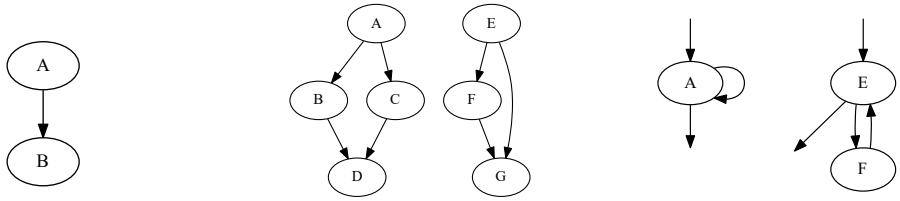
The formal definition from Sabne, Sakdhnagool, and Eigenmann [SSE17] follows a similar approach to their definition of reducible control flow by allowing only a range of structured base patterns which are the only valid outputs from a folding transformation [1] described below. Any other resulting pattern is said to produce unstructured flow.

Definition 7. *A condition node N is **structured** if the region from N to its IPDOM is SESE. The region enclosed by N and its IPDOM is called condition body.*

Definition 8. *If there exists a SESE region (called loop body) between one of the out-edges and in-edges of a loop condition, it is considered a **structured loop condition node**.*

Definition 9. *A condition node is **unstructured** if it is neither a structured condition node, nor a structured loop condition node.*

[1] The AMDGPU target of LLVM currently uses a different technique for structurization.



(a) Sequence (b) Selection (c) Loop

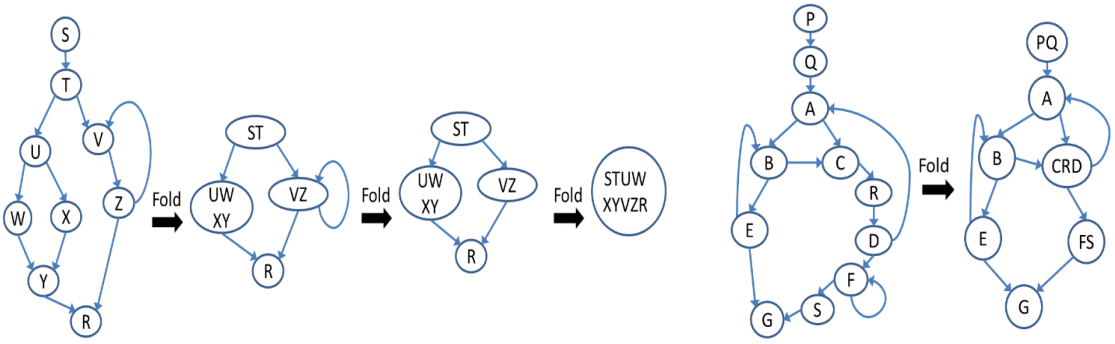
Figure 2.3.: Structured base patterns

The previous three definitions are the basic building blocks for the foldable base patterns:

Definition 10. *The **base pattern of sequence** consists of two nodes A and its only successor B (and their connecting edge $A \rightarrow B$) and A is B's sole predecessor.*

Definition 11. *The **base pattern of selection** consists of a structured condition node, its IPDOM and a non-empty condition body. Any path from the condition node to the IPDOM can have at most one node.*

Definition 12. *The **base pattern of loop** consists of a structured loop condition node and the loop body (with its entry and exit edges) which contains at most one node.*



(a) Folding of a structured CFG (b) Folding of unstructured CFG

Figure 2.4.: Maximal folding algorithm proposed by Sabne, Sakdhnagool, and Eigenmann (Image Source [SSE17]).

Definition 13. *Replacing a base structured pattern with a single node in the CFG is called* ***Folding****, any incoming or outgoing edge is preserved. The maximal folding of a CFG iteratively replaces all structured base patterns until the CFG does not contain any foldable base pattern.*

Definition 14. *If the maximal folding of a CFG contains only a single node, the CFG is said to be* ***structured*** *and unstructured otherwise.*

Sabne, Sakdhnagool, and Eigenmann also show that *structured* CFGs are *reducible* (and hence irreducible CFGs are unstructured) because any maximally folded CFG that contains more than one node must contain at least one unstructured condition node.

Static single assignment form (SSA form) is used in intermediate representations (IR) to simplify analysis and optimization techniques of compiler backends such as LLVM [Bra+13]. SSA form enforces that each variable is defined only once, dominating all its uses. Variables are assigned exactly once but may be used (read) multiple times. Different algorithms exists to generate SSA conformant code from input CFGs, placing phi-functions at nodes where control flow is re-joined, to select one argument (for static assignment) based on how control flow reached this node. Phi-functions are removed during code generation.

3. Related Work

With the rise of programmable GPUs, the problem of divergent control flow on SIMD hardware became apparent. Fung, Sham, Yuan, and Aamodt [Fun+07] describe a technique based on a predicate stack to reconverge at the immediate post-dominator (IPDOM) of a branch. Their algorithm inserts special reconvergence instructions at IPDOMs to operate the stack and re-join the threads as early as possible. For each unique target of a divergent branch, a predicate mask and program counter (PC) is pushed onto the stack. Execution advances based on the top entry of the stack which is popped whenever a reconvergence point (IPDOM) is encountered. Additionally a dynamic cycle-by-cycle regrouping of scalar threads into SIMD warps is proposed to further improve execution unit occupancy.

Diamos, Ashbaugh, Maiyuran, et al. [Dia+11] postulate the concept of a *thread frontier*, that is, the set of threads which branched away from the current warp (in Nvidia terms). Their work explores possible hardware implementations for handling divergent flow in the Sandybridge GPU generation based on existing Per-Thread Program Counters (PTPC) and a new *sorted predicate stack* homed in SRAM or a register file. PTPCs are used to keep track of *thread frontiers* and reconverge threads at the earliest point in time. The compiler assigns a priority to each basic block in the program in reverse post-order (RPOT) which is then enforced by a hardware scheduling policy for executing these blocks in the given priority order. The *thread frontier* for each divergent branch can be statically computed and reconvergence checks are inserted by the compiler to test for disabled threads during execution and reconverge them if possible. In other words, the thread frontier of a basic block is the set of blocks at which dormant threads are waiting to execute. Using this approach, execution of common basic blocks is scheduled such that their re-evaluation is minimized compared to serialized execution. On a conceptual level, the PTPC/Warp PC mechanic is related to the execution mask instruction technique of GCN GPUs. However, it does not allow explicit manipulation of the active threads using bitwise operations because the program counters are advanced implicitly by branch instructions, reducing the space for possible software solutions. The *thread frontiers* implementation based on a new hardware sorted predicate stack is predicted to outperform the implementation based on the existing PTPC of the Sandybridge GPU architecture. Both require additional hardware support while the approach presented

in this thesis only requires executions masks and general purpose bitwise operations (no basic block scheduling in hardware) already present in many architectures.

Compared to OpenMP, MPI, Thread Building Blocks and other multi-core focused programming libraries, Intel's SPMD Program Compiler (**ispc**) [PM12] is inspired by GPU based languages (e.g. inheriting the *uniform* key word) to better utilize SIMD parallelism within a core, leveraging vector units present in modern CPUs. **ispc** implements a C-based single program, multiple data (SPMD) language with a *SPMD-on-SIMD* programming model that allows divergent control flow across SIMD lanes. One of its key features is the ability to call **ispc** routines from C/C++ code and vice versa. **ispc** is designed for CPU architectures, as it assumes a single cache-coherent address space which is not directly applicable to GPU architectures, but an experimental and limited implementation for Nvidia's PTX [Int] exists.

Warp interweaving techniques proposed by Brunie, Collange, and Diamos [BCD12] reclaim SIMD lanes left idle due to divergence, similar to the technique describe by Fung, Sham, Yuan, and Aamodt. *Simultaneous Branch Interweaving* (SBI) does so by co-issuing instructions of divergent branches on the same warp, while *Simultaneous Warp Interweaving* (SWI) fills gaps left by divergence using instructions from other partial warps. Both techniques rely on additional hardware support. SBI mainly improves memory latency hiding by exploiting more memory-level parallelism when the workload of each thread in a warp is balanced (unbalanced case: if a branch is without an else block). SWI utilizes a secondary instruction scheduler, selecting an additional instruction with matching (but not overlapping) execution mask m for the first instruction previously selected by the primary scheduler, making it possible to execute both instructions, possibly from different warps, on the same SIMD group, trading scheduler latency for ALU throughput.

Asanovic, Keckler, Lee, et al. [Asa+13] use convergence and variance analysis in a scalarizing compiler to identify and separate independent scalar operations from parallel code, allowing them to reduce redundant execution and memory access. The key improvements stem from converting thread private registers and vector instructions to shared scalar registers and scalar instructions for as many operations and variables as possible. Additionally, memory loads and stores can be coalesced (for example if the data needed for several threads fits into a scalar register) when an affine analysis pass identifies sequential data access in a warp. Since this work relies on analysis of convergence regions, the question arises whether the reconvergence algorithm proposed in this work could benefit from the scalarization proposed in this paper.

Different software and hardware based solutions for handling divergence of SPMD (Single Program Multiple Data) exist. Lee, Grover, Krashinsky, et al. [Lee+14] evaluate a predication based technique and compare it to an implementation tracking divergence in hardware using a *divergence stack*. Through the course of the paper, the authors explore challenges and tradeoffs of SPMD divergence management architectures and develop *static and runtime branch-uniformity* optimizations that preserve uniform control flow whenever possible. Both methods are related to uniform branch preservation and dynamic scalar branching for empty exec masks used in this thesis, just as the *consensual branches* hardware implementation described in the paper. The latter is used together with a heuristic to decide whether predication gives an advantage over using the divergence stack mechanism. Similar to this thesis, all control flow is removed by predication and linearization of different execution paths to generate a vectorized program. Predicate nodes containing the branch condition mask (and its negation for the *then* branch) are inserted between dependent basic blocks in the control dependence graph (CDG) of the CFG. For the CDG to be thread-aware, *loop*, *continue* and *exit* masks are added to track which threads are executing and exiting the loop at each iteration. Basic blocks are scheduled for linearization using a modified depth-first post-order traversal starting from the exit node in the CFG, visiting the predecessors instead of successors of basic blocks. Finally the authors argue that a masking based software approach is a competitive alternative to managing divergence with hardware, shifting the complexity to the compiler and trading it for a simplified hardware design.

Pohl, Cosenza, and Juurlink [PCJ18] however show that a certain set of hardware capabilities is required to successfully vectorize loops with conditional loads and stores using predication. The NEON SIMD extension for ARM lacks support for masked memory operations, the authors propose runtime checks to ensure all load an store operations within loops are legal. Their evaluations shows that compared to AVX2 (which supports predicated loads and stores) some additional overhead is introduced to mitigate the lack thereof and not all cases of divergent control flow within loops can be vectorized with this approach. A survey shows that both LLVM and GCC failed to vectorize loops for ARM NEON in cases where they emitted SIMD instructions for Intel CPUs supporting the AVX2 extension.

The PTX and Ocelot based linearization method proposed by Anantpur and R. [AR14] structurizes CFGs without duplicating user code but adds a substantial number of predicate variables to linearize divergent control flow. This in turn increases register pressure due to an additional block (containing the inserted guard-variable) for each basic block in the original CFG. An additional optimization pass is introduced to remove unnecessary (nested) guard checks to improve the situation, but the overhead

problem remains with possibly more than 10% increased number of registers and up to 27% more basic blocks. Apart from the register allocation problem, existing uniform control flow is not preserved and unnecessarily linearized, further impairing possible performance improvements. This is not a feasible solution for AMD GCN based GPUs where scalar registers are scarce resources with strong influence on performance.

Ren, Krishnamoorthy, Agrawal, and Kulkarni [Ren+17] present a framework for scheduling *task blocks* to vector units or (SMT) threads, allowing better utilization for programs with both data and task-parallel workloads. In modern CPUs, each core has a dedicated SIMD hardware suited for data-parallel *task blocks* while also being able to execute several *task blocks* at the same time on a multicore chip. In contrast to **ispc**, the *grain free* scheduler presented in this paper allows to independently schedule each *task block*, better exploiting available nested parallelism. A simple new programming language is used to express parallel tasks nested within data-parallelism.

Moll and Hack [MH18] present a partial control flow linearization (if-conversion) technique with provable guarantees on the preservation of uniform flow. The algorithm processes basic blocks of the CFG in *block index order*, a traversal of the dominator-tree, resulting in contiguous dominance regions where the previous block dominates the following blocks in the ordering. This ordering approach is related to the proposed CFG traversals respecting post-dominance relations (i.e. DFPD) discussed in section 4.3.1. The main idea here is to if-convert a divergent block B by giving it only a single outgoing edge to a selected block $Next$ in the resulting linearized CFG where each remaining successor of B in the original CFG post-dominates B in the output CFG. Finally, the BOSCC (Branch on Supercondition Code) technique is used to skip over regions where static analysis failed to prove uniformity and the predicate of the region entry evaluates to *false* for all SIMD threads. In contrast to the lowering method proposed in this work, a BOSCC gadget (containing instructions similar to the execution mask operations) is inserted into the CFG *before linearization* instead of after the reconvergence transformation. In comparison, scalar branches inserted in our lowering method are not only added to handle dynamically uniform flow, but also for correctness due to possible side effects of GCN vector instructions with empty execution masks. *Partial Linearization* and the algorithms proposed in this work have similar guarantees with respect to uniform control flow. *Partial Linearization* operates on unstructured input CFGs and does not insert additional branches or code duplicates.

Many of the related publications recognize the significance of post-dominance for divergence management, even the early programmable GPUs implemented divergence stacks using properties of immediate post-dominators [Fun+07] but none seem to transfer this property to control flow graphs as proposed by this thesis.

4. Reconverging control flow graphs

Vectorization of divergent control flow is mainly done to better utilize SIMD units. Transformations doing so can require reducible or structured control flow [MH18]. Ideally a vectorising algorithm should work with irreducible control flow and retain uniform (scalar) branches without replicating code. Common steps for vectorization (if conversion) based on divergence analysis are [MH18] [Chi17] [AR14]:

1. Each possible path of a branch is executed
2. Insert instructions to predicate control flow for basic blocks (masking)
3. Replace non-uniform branch instructions with vector instructions
4. Results are conditionally selected
5. Remove divergent branches from the CFG

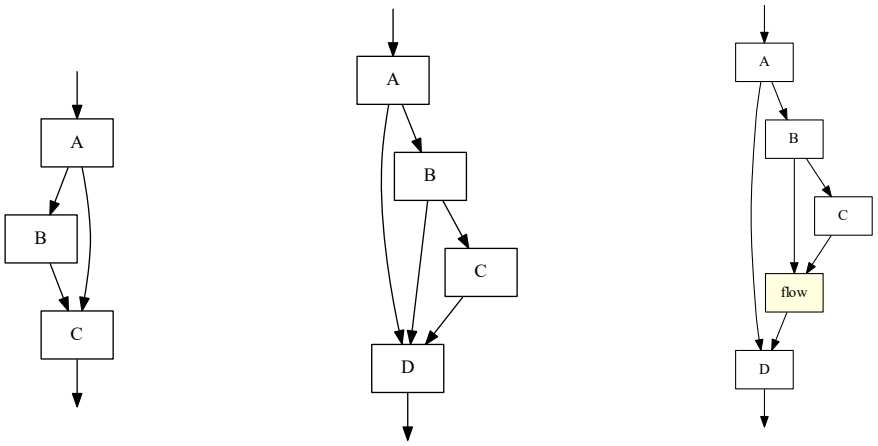
In the context of GPUs, certain constraints on the input program may apply: Recursion and infinite loops are not supported by shading languages [FLB; Rui], meaning that loops are required to have reachable exits. For the proposed algorithm, CFGs are required to only have a single entry and a single exit node. Nodes that are unreachable from the entry node will not be considered. Conditional branch nodes with more than two successors (e.g. switch statements) are assumed to be resolved to ordinary conditional branch nodes (with two successors) by an earlier transformation.

4.1. Solution outline

When looking at the problem from the perspective of what is actually required to lower control flow to wave level (ISA code), we can reduce the requirements by capturing thread divergence: whenever threads diverge at a non-uniform branch, a single point in the CFG needs to exists where those threads reconverge. Hence follows the definition of a reconverging CFG:

Definition 15. *A control flow graph is reconverging if every non-uniform condition node (terminator T) has exactly two successors, one of which post-dominates it (primary successor).*

Listing 4.1 shows that the wave will first branch into the secondary successor. Because the primary successor post-dominates it, the threads left behind by the wave can be re-joined or re-converged at this point.



(a) Reconverging subgraph of a CFG. C is the primary successor and reconvergence point of threads that did not branch to B.

(b) Example of reconverging CFG with multiple branches sharing the same primary successor at node D.

(c) Initializing the rejoin-mask to 0 can be avoided by inserting an additional flow block with B and C as predecessors.

Figure 4.1.: Base case of a reconverging conditional branch.

The algorithm for vectorizing divergent control flow using the properties of reconverging CFGs consists of the following steps:

1. Make sure a unique sink exists by adding a common virtual exit to be able to merge control flow from non-uniform branches that lead to different exit nodes.

2. Determine an ordering of the basic blocks based on the topology of the CFG. The correctness of the following transformation is not affected by the ordering created in this step, but the quality of the resulting CFG depends on it.

3. Transform the control flow to ensure existence of reconvergence points by adding **flow** blocks and rerouting edges accordingly.

4. Inject predication operations to lower the reconverging CFG to wave-level. Optionally keep scalar branch instructions to skip blocks that would be executed with an empty execution mask.

4.2. Lowering a reconverging CFG to wave-level control flow

To motivate why reconverging CFGs help vectorizing divergent control flow, we will first explore how to exploit the reconvergence property to lower thread-level to wave-level control flow before looking at the transformation required to generate reconverging CFGs from arbitrary CFGs.

Listing 4.1: Pseudo GCN ISA predication for control flow of Figure 4.1b

```
1  A:  // code for A ...
2      v_cmp_???       s[0:1], ...          // condition for jumping to primary successor D
3      s_mov_b64       s[2:3], 0            // initialize re-join mask of B to 0
4      s_andn2_b64     exec, exec, s[0:1]   // subtract re-join mask from execution mask
5      s_cbranch_execz D                    // branch to D if execution mask is empty
6
7  B:  // code for B ...
8      v_cmp_???       s[2:3], ...          // condition for jumping to D
9      s_andn2_b64     exec, exec, s[2:3]   // subtract re-join mask from execution mask
10     s_cbranch_execz D                    // branch to D if execution mask is empty
11
12 C:  // code for C ...
13 D:  s_or_b64        exec, exec, s[0:1]   // add threads that did not branch to B
14     s_or_b64        exec, exec, s[2:3]   // add threads that did not branch to C
15     // code for D ...
```

The example in Listing 4.1 shows a simple approach for lowering several nested branches that share a common primary successor. The idea is to store the bit mask of threads that will branch to the primary successor in separate registers for each divergent condition node and later re-join the threads at the beginning of the primary successor by accumulating the saved bit masks from said registers in the execution mask register. This approach requires additional scalar registers to store the respective *re-join bitmask* for each non-uniform branch. However, the same control flow predication can be achieved with just one virtual register m by directly accumulating diverging threads in m. This requires that m is correctly initialized at entry points to the divergent flow. For example the re-join mask must be initialized to 0 (see Listing 4.1), so that merging it into the execution mask at the shared primary successor does not corrupt the execution mask with unrelated data still residing in the register.

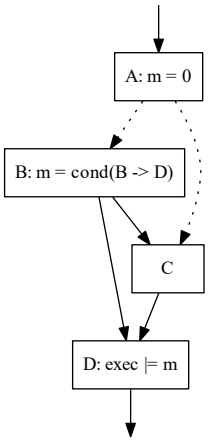
The basic steps for this improved approach are: For every non-uniform branch, identify the primary and secondary successors. Add instructions to each condition node to store (or add) the *re-join bitmask* containing the threads that branch to the primary successor to *m* and subtract (bitwise AND-NOT) it from the execution mask (*exec*). Optionally append a conditional scalar branch instruction that jumps to the primary successor in the case where the remaining execution mask is empty. At the top of the primary successor's basic block, insert a single bitwise OR instruction to merge the final *re-join mask* for all predecessors. The additional scalar branch increases performance by skipping instructions which are dormant with an empty execution mask. This optimization is also required for correctness on some architectures where vector instructions have side effects (e.g. setting flag registers) even with an empty execution mask.

The lowering algorithm operates on three different base patterns occurring in reconverging CFGs. The base case handles perfectly nested non-uniform control flow as seen in Figure 4.1b. Because the definition of reconverging control flow is only concerned with the outgoing edges of non-uniform nodes, incoming uniform edges to any of the nested nodes of the base pattern may exist. These incoming edges (compare to Figure 4.2a) might skip instructions of previous basic blocks which initialize the re-join mask. Therefore the source node of the incoming edge has to be fixed by initializing the re-join mask. Furthermore, non-uniform reconverging subgraphs can be nested in uniform control flow, sharing the same primary successor (as seen in Figure 4.3a) where uniform and non-uniform control flow is mixed. Similarly, the case of a incoming backward edge from the primary successor has to be treated by the algorithm.

Listing 4.2: Improved pseudo GCN ISA predication for control flow of Figure 4.1b

```
1    A:  // code for A
2        v_cmp_???       s[0:1], ...          // initialize re-join mask in m
3        s_andn2_b64     exec, exec, s[0:1]   // subtract re-join mask stored in s[0:1]
4        s_cbranch_execz D
5
6    B:  // code for B
7        v_cmp_???       vcc, ...             // condition for jumping to D
8        s_or_b64        s[0:1], vcc          // accumulate re-join mask in m
9        s_andn2_b64     exec, exec, vcc      // subtract re-join mask from execution mask
10       s_cbranch_execz D
11
12   C:  // code for C
13   D:  s_or_b64        exec, exec, s[0:1]   // add previously subtracted re-join masks
14       // code for D
```

To show that the reconvergence property is enough to generate wave-level control flow from a CFG, the last step of the algorithm is outlined below. It handles uniform and non-uniform control flow in a unified manner.



(a) Nested reconverging CFG with incoming uniform edges.

Set	**Elements**
P	B
P^*	B, C, D
P_0	B
$P \setminus P_0$	$\emptyset$
$P^* \setminus P_0$	C, D
Q	A

(b) Set elements for the CFG in Figure 4.2a for node D.

Figure 4.2.: Example application of the lowering algorithm with incoming uniform edges from node A.

The starting point of the algorithm is the primary successor B of a divergent branch. We consider P the set of predecessors for which B is the primary successor and P^* the set of nodes which are reachable from these predecessors without going through B (capturing secondary successors and B itself). P_0 is the subset of P which contains those predecessors that do not themselves have predecessors in P^*. That is, P_0 is the set of sources in the subgraph induced by P^*. Finally we consider the set Q of predecessors of the reachable set P^*, excluding the entry nodes $\in P^* \setminus P_0$, which are not in the reachable set P^*, meaning all the 'extended' entry nodes which are not in the reconverging subgraph. The sets P_0 and Q cover all nodes entering to the flow of a particular reconverging subgraph. See Figures 4.2, 4.3 and 4.4 for examples.

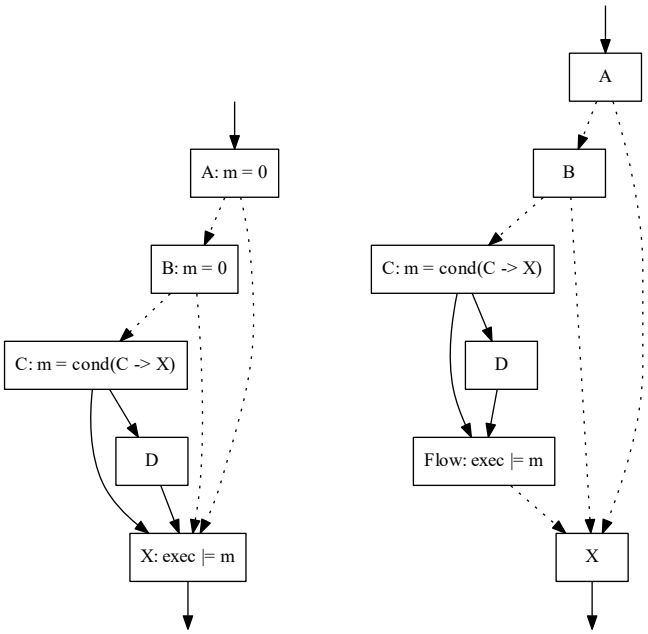
LowerReconvCFG(CFG)

```
1  for each unprocessed primary successor B ∈ CFG
2      // get predecessors for which B is the primary successor
3      P = {x : x ∈ B.predecessors ∧ x.primarySuccessor = B}
4      // get set of nodes that are reachable from any node in P
5      P* = {x : ∃y ∈ P and a simple path from y to B that contains x}
6      // get nodes of P which do not have predecessors in P*
7      P_0 = {x : x ∈ P ∧ x.predecessors ∩ P* = ∅}
8      // m is a virtual register reserved to hold the re-join mask for B
9      for p ∈ P
10         if p ∈ P_0 // handle entry nodes
11             // set m to the mask of threads which branch to B
12             Insert MOVE instruction in p to set m = cond(p → B)
13         else // handle nested nodes (p ∈ P \ P_0)
14             // add the mask of threads which branch to B to m
15             Insert bitwise OR instruction in p to set m = m ∨ cond(p → B)
16         Insert bitwise AND-NOT instruction in p to subtract cond(p → B) from exec
17         Insert scalar branch instruction in p to skip to B if exec register is empty
18     // get predecessors of P* \ P_0 which are not in P*
19     Q = {x : x is a predecessor of some y ∈ P* \ P_0 ∧ x ∉ P*}
20     // handle nodes with incoming uniform branches or backward edges
21     for p ∈ Q
22         // Initialize the re-join mask m to 0
23         Insert MOVE instruction in p to set m = 0
24     Insert bitwise OR instruction in B to add the re-join mask m to exec
25     B.processed = true
```

The LowerReconvCFG algorithm will recombine the set of disjoint threads ($\in P \setminus P_0$) with a bitwise OR instructions inserted at Line 15 (given that the input CFG is reconverging). The lines 12 and 23 make sure that all paths entering P^* initialize the re-join mask register correctly. The resulting wave-level control flow of this algorithm is equivalent to the input thread-level control flow.

Figure 4.2 shows the case of a reconverging CFG with incoming uniform edges from node A. The algorithm correctly initializes the re-join mask m to 0 to make sure the execution mask will not be corrupted by adding m when the edge from A to C is taken. In the case of nested uniform control flow as seen in Figure 4.3a, this leads to unnecessary re-initialization of m in node B although it is dominated by A (which already initialized the re-join mask correctly). Changing the algorithm to skip

initialization in this edge case would impair the simplicity of this unified approach while also changing the register lifetime which might result in increased register pressure in B. Optionally, a new flow block (see Figure 4.3b) with predecessors C and D could be introduced to detach the reconverging pattern from the incoming uniform control flow at X. In this case, X would no longer be the primary successor for C and the algorithm would instead process the newly created flow block.



(a) Reconverging sub CFG with incoming uniform edges.

(b) Reconverging sub CFG with flow block.

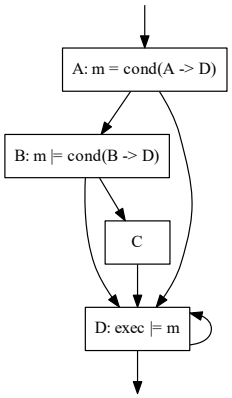
Set	Elements
P	C
P^*	C, D, X
P_0	C
$P \setminus P_0$	$\emptyset$
$P^* \setminus P_0$	D, X
Q	A, B

(c) Set elements for the CFG in Figure 4.3a for primary successor X.

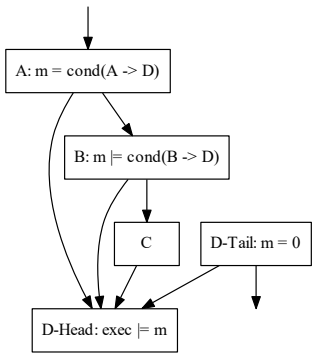
Figure 4.3.: Example application of the lowering algorithm with incoming uniform and non-uniform edges from nodes A and B.

The algorithm described above requires an additional step to handle incoming backward edges (for example from loops at the primary successor) because the re-join mask is only initialized to the condition bits for branching from the entry node to

the primary successor but not from the incoming backward edge. This requires the re-join mask to be nulled. For the sake of simplicity we don't want to change how the algorithm operates on the given sets of nodes. Therefore the primary successor needs to be altered in way such that it becomes a member of set Q and m is initialized to 0. A new subroutine SplitNode is invoked on the primary successor before constructing the node sets. It performs a virtual split on the input node B, creating a B_{head} node with all the original predecessors of B (including the backward edge B_{tail}) and a B_{tail} node containing the successors of B. Because B_{tail} is now a predecessor of B_{head} which is not in the reachable set P^*, it becomes a member of Q (see Figure 4.4) and is properly initialized while B_{head} stays the unique sink of P^*, merging the re-join mask m into the *exec* register. In practice the code of B_{tail} directly follows after B_{head}.



(a) Before splitting the primary successor.



(b) After splitting the primary successor.

Set	Elements
P	A, B
P^*	A, B, C, D-Head
P_0	A
$P \setminus P_0$	B
$P^* \setminus P_0$	B, C, D-Head
Q	D-Tail

(c) Example set elements for the CFG in Figure 4.4b for node D.

Figure 4.4.: Virtual node splitting example.

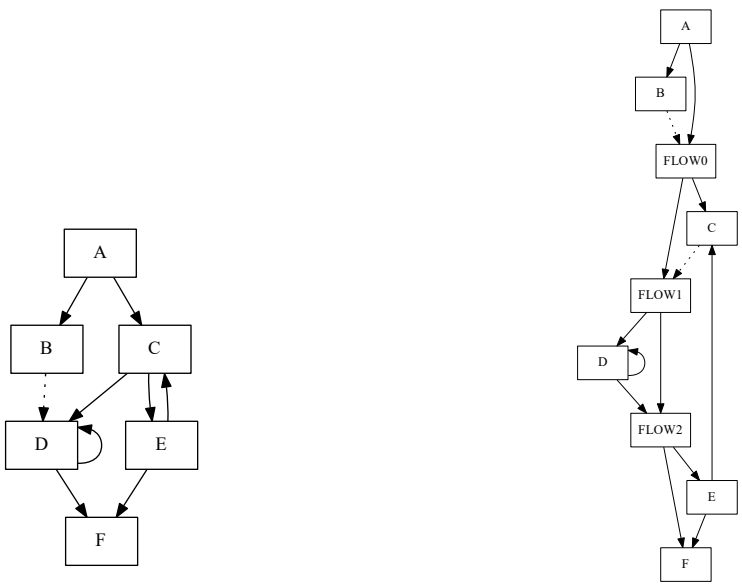
Similar to the case of mixing incoming uniform flow (as seen in Figure 4.3), it is possible to introduce a *flow* block (with predecessors B and C) to re-join divergent threads before entering the original primary successor at node D (compare to Figure 4.1c). Inserting a *flow* block dominating the primary successor (with its reachable predecessors from P^*) is a simple and valuable optimization of the algorithm, although there are cases (see Figure 4.2a) where the initialization to 0 is still required.

4.3. Transforming to reconverging control flow

This section is concerned with transforming any input CFG to a reconverging one following Definition 15. The algorithm is split into two phases:

1. Computing an order of nodes in which the basic blocks are added to the *open tree*.
2. Maintaining an *open tree* structure used to reroute control flow such that reconvergence points exists.

Any ordering of basic blocks is valid if the unique sink or exit block comes last. For CFGs with multiple exit blocks, a new basic block which post-dominates these blocks is added to the CFG to create a unique sink. Different strategies based on depth-first, breadth-first and reverse post-order traversal of the CFG will be discussed later in Section 4.3.1. The ordering affects the runtime of the algorithm as well as the quality of generated control flow. To be able to better reason about the ordering we will first discuss the main pass maintaining the *open tree* structure OT.



(a) Unstructured input CFG. (b) Reconverging CFG.

Figure 4.5.: Input CFG before and after transformation.

The nodes of OT correspond to visited nodes of the ordering, resulting in a single pass over the CFG. Each node (except the virtual root node) has a number of open control

flow edges, that is, edges taken from the input CFG which have not yet been processed. These edges are categorized in two sets: *Incoming* and *Outgoing* edges, i.e. the sources and targets of the node. Once all open edges are closed by processing the target blocks of those edges, the node can be removed from the tree, moving its successors to the parent node. However, to make the following analysis more comprehensible, closed nodes are handled as if they were not removed from the tree to improve the traceability of each step.

The direction of an edge with respect to the ordering depends on the position source and target within the ordering. An edge is directed forward if the target block comes after the source block in the ordering, otherwise it is considered backwards. Therefore an edge (B, B) is considered backwards.

(a) Open edges contained in subtree routed at B.

(b) Open edges contained in common subtree of B and C.

Figure 4.6.: Open edges in the OT can be either incoming backward edges or outgoing forward edges because all predecessors B must be ancestors of the OT node according to the *invariant*. Visited nodes have a black frame, unvisited nodes light grey. Solid edges symbolize OT parent-child edges. Dotted edges are open outgoing, dashed edges are open incoming edges.

The proposed algorithm will maintain the following **invariant**:

Definition 16. *For every open outgoing edge E = (B, X) of a* visited *basic block B the following holds: Each open outgoing Edge E′ = (C, Y) of a visited basic block C which can be reached from B through visited basic blocks in the CFG, is either contained in the subtree rooted at B, or the smallest common subtree containing both B and C only has open outgoing edges to $X = Y$. The latter is called* relaxed subtree condition.

The objective of the following subroutine is to ensure that any node B added to the OT will become the descendant of all its predecessors. AddBasicBlockToOT will add the new node for basic block B as a child to its unique predecessor in the OT. If there is no predecessor in the structure, B is added to the *root* node. If there are multiple predecessors, InterleavePaths collapses all paths from the *root* leading to B into a single path containing all of its predecessors so that a unique lowest predecessor exists and B can be attached. After adding the node to the OT, B is considered *visited* and edges from its predecessors can be closed.

AddBasicBlockToOT(B)
1 $P = \{x : x \in B.incoming \wedge x.visited = \text{TRUE}\}$
2 // Collapse all paths leading from the root to B into a single path
3 $p =$ InterleavePaths(P)
4 Add B as a child to p.
5 **for** $pred \in P$
6 CloseEdge(pred, B)
7 $B.visited = \text{TRUE}$

The path interleaving algorithm first ascends to the nearest common ancestor of B's visited predecessors to find the starting point and then descends the structure to the leaf nodes, concatenating the traversed branches into a single path. This makes sure that B becomes a descendant of all its predecessors and adding a node with non unique predecessors to the OT maintains its invariant. AddBasicBlockToOT maintains the invariant when adding a basic block without outgoing backward edges. This assertion is based on the fact that all predecessors of B are ancestors of B and InterleavePaths will not change any common subtree S that is relevant for the relaxed subtree condition, except when adding the unique unvisited successor of S.

Definition 17. *A node of the OT is called* armed *if the terminator of the corresponding basic block is divergent and one of the outgoing edges has already been closed.*

The Reconverge subroutine outlines the steps necessary to identify and resolve control flow constructs which prevent reconvergence of a CFG as described by Definition 15. The main idea (implemented by Reroute) is to split control flow of *critical* nodes (e.g. armed for condition 1) into two outgoing edges: The first one will become the primary successor and the second one the secondary successor. The target of the second edge is the first *visited* successor of the node, while the first edge leads to a newly created *flow* node which post-dominates the *critical* node and the visited successor.

```
RECONVERGE(O)
 1  Add virtual root node to OT
 2  for B ∈ O // Process basic block B of ordering O
 3      P = {x : x ∈ B.incoming ∧ x.armed = TRUE}
 4      // Let S be the set of subtrees rooted at nodes in P.
 5      S = BUILDSUBTREES(P)
 6      S_o = {x : x ∈ S ∧ B ∉ x.outgoing}
 7      // Reroute open outgoing edges of nodes in S, not leading to B
 8      if S_o ≠ ∅ // Condition 1
 9          REROUTE(S)
10      ADDBASICBLOCKTOOT(B)
11      M = {x : x ∈ B.outgoing ∧ x.visited = FALSE}
12      N = {x : x ∈ B.outgoing ∧ x.visited = TRUE}
13      // B has unvisited successors and visited successors different from B
14      if M ≠ ∅ ∧ (N \ B) ≠ ∅
15          // Let S be minimal subforest rooted at B and nodes in N.
16          S = BUILDSUBTREES(N ∪ B)
17          S_o = {x : ∃y ∈ S ∧ x.outgoing \ y.outgoing ≠ ∅}
18          // Reroute S if it has multiple roots
19          // Or open outgoing edges to multiple targets.
20          if S_o ≠ ∅ ∨ |x.roots| > 1 // Condition 2
21              REROUTE(S)
22      for s ∈ N
23          CLOSEEDGE(B, s) // Close edge to visited successor.
```

When processing a node B of ordering O there are two situations in which control flow needs to be rerouted: The first one makes sure that open *incoming* edges do not break the invariant before adding node B to the OT. The second one maintains the invariant for open *outgoing* edges of B after it has been added to the structure. Condition 1 shown in Line 8 identifies incoming *armed* flow (which provides the closed successor needed for the split) where the subtrees routed at these *armed* nodes have open outgoing edges not leading to B itself. Condition 2 (Line 20) captures the case where B has *unvisited* successors and *visited* successors different from B and the minimal subforest rooted at visited successors of B and B itself contains multiple roots (nodes which are not descendants of other subtrees) or multiple open outgoing edges to multiple targets.

```
REROUTE(S)
1  Create new basic block X
2  // For each node N of the set of subtrees S.
3  for N ∈ S
4      // For each edge from N to target B of the open outgoing edges of N.
5      for E = (N, B) ∈ N.outgoing
6          Replace E by the edge (N, X)
7          Add an edge (X, B) if it does not already exist.
8  ADDBASICBLOCKTOOT(X)
```

REROUTE operates on open outgoing edges of nodes in the set of subtrees S which were constructed under either Condition 1 or 2. Each of these edges $E = (A, B)$ is rerouted through a new *flow* basic block X by replacing edge E with two new edges, one leading into the *flow* block (A, X) and one leading out to the original target (X, B) if it does not already exist. Like any other basic block, X is added to the OT through ADDBASICBLOCKTOOT(X).

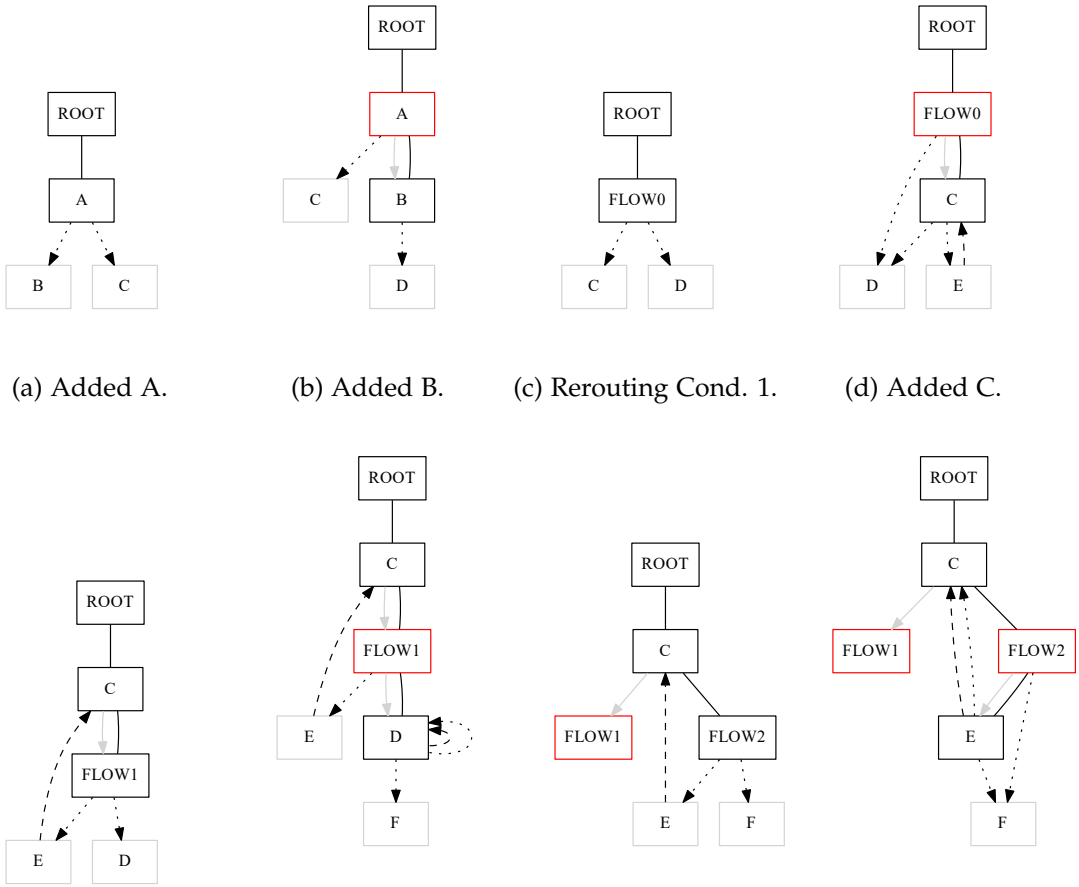
Lemma 1. *Adding the flow block to the OT in* REROUTE *preserves the invariant.*

Flow basic blocks added by the rerouting subroutine invoked in Line 9 and 21 maintain the open tree invariant because they lack outgoing backward edges, i.e. edges that might lead out of the current subtree.

Lemma 2. *Adding the basic block B to the OT in Line 10 preserves the invariant.*

When adding the current basic block B in Line 10, after the subtrees S have be rerouted under condition 1, the open outgoing edges reachable through Bs backward edges (i.e. edges to previously visited blocks) are attached to OT ancestors of the inserted *flow* block and backward edges are closed by ADDBASICBLOCKTOOT(flow). Therefore the invariant when S was rerouted under same reasoning as adding *flow* blocks. When S contains subtree nodes with open outgoing edges, but is not rerouted under condition 1, the invariant holds because of the relaxed subtree condition.

The algorithm produces reconverging control flow from arbitrary input CFGs because closing the remaining outgoing edge E of an *armed* basic block B makes the target of edge E the post-dominator of B. This follows from the *invariant* and the fact that all open outgoing edges in the subtree routed at B are closed at the same time, pointing to the same successor. However, rerouting under said conditions is only possible if any critical block B is closed before the exit block such that a path in OT through remaining open outgoing edges to the exit node exists and the target of E can become a post-dominator. As a consequence, the exit block must be processed last by RECONVERGE.



(a) Added A. (b) Added B. (c) Rerouting Cond. 1. (d) Added C.

(e) Reroute Cond. 1 (f) Added D. (g) Reroute Cond. 1. (h) Added E.

Figure 4.7.: Stages of the OT while processing CFG shown in Figure 4.5a in breadth-first order: $A \to B \to C \to D \to E \to F$. Closed nodes are removed from the tree. Adding basic blocks *Root* and *F* is omitted because it closes all remaining edges, resulting in OT only containing the root node. Red nodes are armed. Solid light grey edges are closed. Visited nodes have a black frame, unvisited nodes light grey. Solid edges symbolize OT parent-child edges. Dotted edges are open outgoing, dashed edges are open incoming edges.

The Reconverge algorithm will not alter control flow that is already reconverging given that the input ordering obeys post-dominance. This follows from the fact that

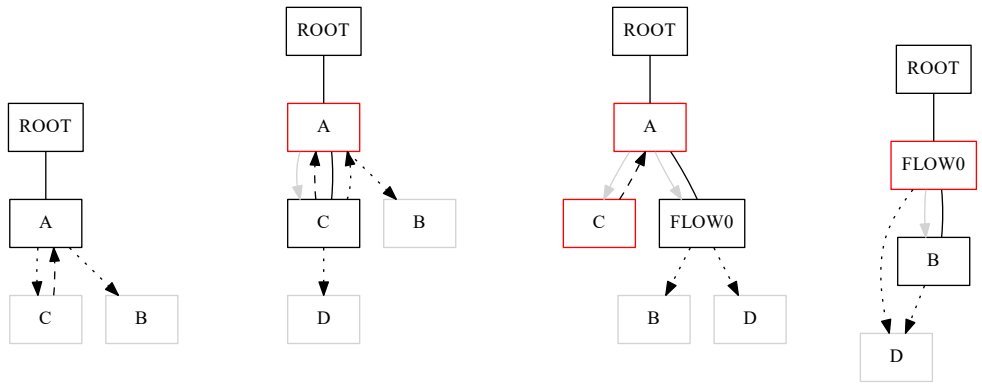
condition 1 and 2 only reroute nodes inflicting the reconvergence definition and no artificial backward edges are introduced by the ordering. For condition 1, if the subtree routed at *armed* (and therefore divergent) predecessors P of a node B contains at least one open outgoing edge that leads to a different node than B, a successor of the *armed* predecessor exists which is not a post-dominator (primary successor) of the armed predecessor. Looking at open tree structure of Figure 4.7b (of input CFG (Figure 4.5a) it can be seen that an open edge from the *visited* node B to D exists and none of the successors of the armed node A post-dominate.

(a) Example CFG with block C inflicting reconvergence.

(b) Reconverged CFG of input shown on the left.

Figure 4.8.: Example input ant output CFG for rerouting under condition 2 using ordering $A \to C \to B \to D$.

For condition 2, if the subforest S rooted at a node B and its visited successors (i.e. the targets of *visited* successors) contains multiple roots with open outgoing edges or open multiple open outgoing edges to different basic blocks, S is rerouted. This is necessary because the successors of targets of backward edges do not post-dominate B if they have multiple open outgoing edges. Node A of the CFG shown in Figure 4.8a has no primary successor because C has an edge to exit node D instead of B which would make B the post-dominator of A. When looking at Figure 4.9b we can see that node A is armed *after* C has been added and condition 1 fails to capture critical backward edges of C because it is only concerned with armed predecessors, condition 2 resolves the remaining critical nodes.



(a) Added A. (b) Added C. (c) Rerouted Cond. 2. (d) Added B.

Figure 4.9.: Steps of the open tree algorithm for CFG shown in Figure 4.8a using ordering $A \rightarrow C \rightarrow B \rightarrow D$. Closed nodes are removed from the tree. Steps for adding *Root* and *D* are omitted. Red nodes are armed. Solid light grey edges are closed. Visited nodes have a black frame, unvisited nodes light grey. Solid edges symbolize OT parent-child edges. Dotted edges are open outgoing, dashed edges are open incoming edges.

4.3.1. Basic block ordering

(a) Reconverging input CFG.

(b) Transformed reconverging input CFG using order $A \rightarrow C \rightarrow B \rightarrow D$.

Figure 4.10.: Choice of ordering influences quality resulting CFG.

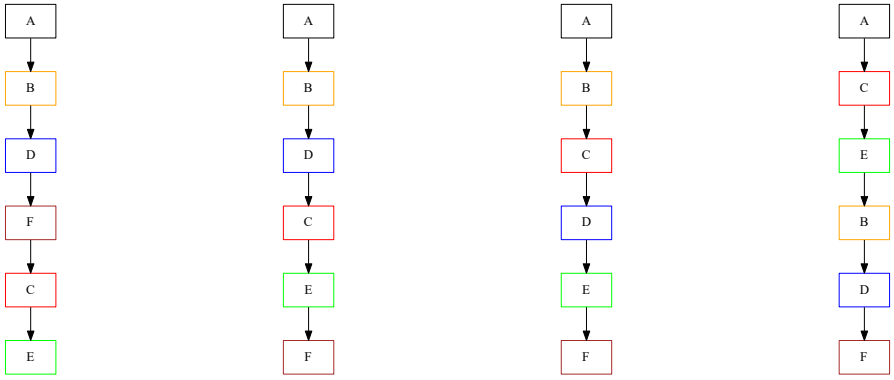
Any ordering of basic blocks produced by traversing the original CFG is required to uniquely contain each reachable basic block and that the exit block comes last. The example of conventional depth-first traversal shown in Figure 4.11a does not produce a valid ordering because F is not the last block. All ordering methods discussed in this section are modified to produce valid inputs for the RECONVERGE subroutine.

The proposed algorithm implements a traversal of the CFG which obeys the post-dominance relations of basic blocks. The choice of ordering in which basic blocks are processed directly influences where *flow* blocks are placed in the CFG and how the original edges are rerouted. When processing the already reconverging CFG shown in Figure 4.10a in forward order $A \rightarrow C \rightarrow B \rightarrow D$ the REROUTE subroutine introduces an unnecessary backward edge from block B to C (Figure 4.10b). This behavior can be avoided by first processing the basic block which does not post-dominate the block containing the branch instruction.

The COMPUTEORDERING subroutine will traverse the input CFG in a depth-first (**DF**) manner but any post-dominator of a block should be positioned after its predecessors in the ordering. This is done by deferring traversal of a unvisited block B if a unvisited successor S of a ancestor of a previously *visited* block A is post-dominated by S. When there is more than one unvisited successor of block A, PICKBLOCK chooses the one which is not a post-dominator of A. Under this modification, the traversal implemented in COMPUTEORDERING will be called *Depth-first with Post-Dominance* (**DFPD**).

```
ComputeOrdering(EntryBlock)
 1  Ordering = ∅
 2  A = EntryBlock
 3  ToVisit = {A}
 4  while ToVisit ≠ ∅
 5      Ordering = {Ordering ∪ A}
 6      ToVisit = {ToVisit \ A}
 7      for B ∈ A.successors
 8          if B ∉ Ordering ∧ B ∉ ToVisit
 9              ToVisit = {ToVisit ∪ B} // Add discovered successor
10      if ToVisit = ∅
11          return Ordering
12      Next = nullptr
13      for B ∈ A.successors ∧ B ∉ Ordering
14          Next = B
15          for P ∈ A.predecessors ∧ P ≠ A
16              for S ∈ {s ∈ P.successors | s ∉ Ordering}
17                  if B == S ∨ PostDominates(B, S)
18                      Next = nullptr // Reject traversal of B
19                      goto Line 13
20      if Next == nullptr
21          Next = PickBlock(ToVisit, A)
22      A = Next
23  return Ordering
```

Another category of orderings is based on breadth-first (**BF**) traversal of input CFGs. The underlying idea is to traverse basic blocks with the shortest path length from the function entry block (root) first. However this is not enough to reflect dominance relations in the resulting ordering because incoming edges from untraversed subgraphs can exist. Therefore, a working set of unvisited successors (sorted by distance to the root) is used to select the next block to be traversed. If a block has two successors, select the one that does not post-dominate first (similar to PickBlock). In the case that all of the blocks in the working set have unvisited predecessors, the first block of the working set that is not the sink of the CFG is selected as a tie-breaker. The resulting ordering will be referred to as *Breadth-first with Post-Dominance* (**BFPD**).



(a) Depth-first. (b) DFPD. (c) Breadth-first. (d) RPOT.

Figure 4.11.: Example orderings created by traversal of the CFG shown in Figure 4.5a.

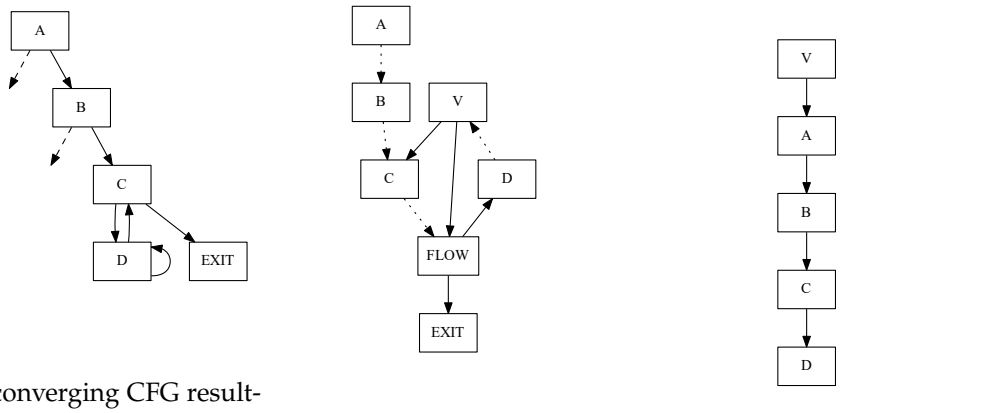
In *reverse post-order traversal* (**RPOT**, topological sort) a block is only appended to the front of the ordering once all its successors have been *visited*. This guarantees that any block in the ordering appears before its successors in the CFG and therefore maintains dominance relations.

Moll and Hack [MH18] describe a traversal of basic blocks maintaining *dominance regions* (**DomReg**) when enumerating nodes of a CFG. Each block in the output ordering is followed by the blocks it dominates, forming a continues stream of blocks which can be gathered from branches of a dominator-tree. A depth-first traversal of the CFG's dominator-tree yields a suitable output ordering if the lowest branch containing the exit node is traversed last.

Prepare ordering

Before passing the ordered set of blocks to the Reconverge algorithm, a preparatory pass called PrepareOrdering makes sure that the exit block comes last and no divergent block has two backward edges. For the first requirement, the exit block is removed from its location within the ordering and appended to the end so that blocks in-between are shifted to their respective predecessor's position. For the second requirement, PrepareOrdering modifies non-uniform terminators with two outgoing backward edges such that the two backward edges are converted to one backward edge and two forward edges with respect to the ordering. It does so by inserting a new virtual node Y for each block B with a non-uniform terminator where both outgoing

edges are backwards relative to its position in the ordering. Y is inserted above the target nodes of the outgoing backward edges and the terminator of B is replaced by an unconditional branch to Y and the original terminator is moved into Y. Y is added to the ordering solely for the purpose of reducing the number of backward edges and does not need to be added to the underlying CFG. This is done because nodes with one or more open outgoing edges to *visited* predecessors in the OT will be processed by condition 2 and the chance that two nodes have multiple open outgoing edges or multiple roots within the subtree is greater with two nodes. This results in a higher number of inserted *flow* blocks. Unlike the *flow* node, a virtual node does not need to be added to the actual CFG, so reordering the input order is preferable.



(a) Reconverging CFG resulting in ordering with two backward edges at node D.

(b) Reconverged CFG of order: $V \rightarrow A \rightarrow B \rightarrow C \rightarrow D$.

(c) Virtual node V inserted before A.

Figure 4.12.: Choice for inserting virtual nodes V into the ordering.

If all nodes of the ordering are already reconverging and it contains nodes with multiple backward edges, PrepareOrdering will unnecessarily add virtual nodes to the ordering. This can cause the Reconverge routine to reroute through additional *flow* blocks even though the original CFG is already reconverging. Therefore either before executing the complete algorithm or before PrepareOrdering, a reconvergence check on the input CFG is required to avoid transforming the CFG again.

Consider the partial example CFG shown in Figure 4.12 and the corresponding node ordering $A \rightarrow B \rightarrow C \rightarrow D$. The only requirement for inserting the virtual node V into the ordering is that it comes before the critical nodes successors. Several possibilities arise as V can be inserted at any position before node C to resolve the backward edges. As a convention virtual nodes are inserted at the front of the ordering.

Comparison

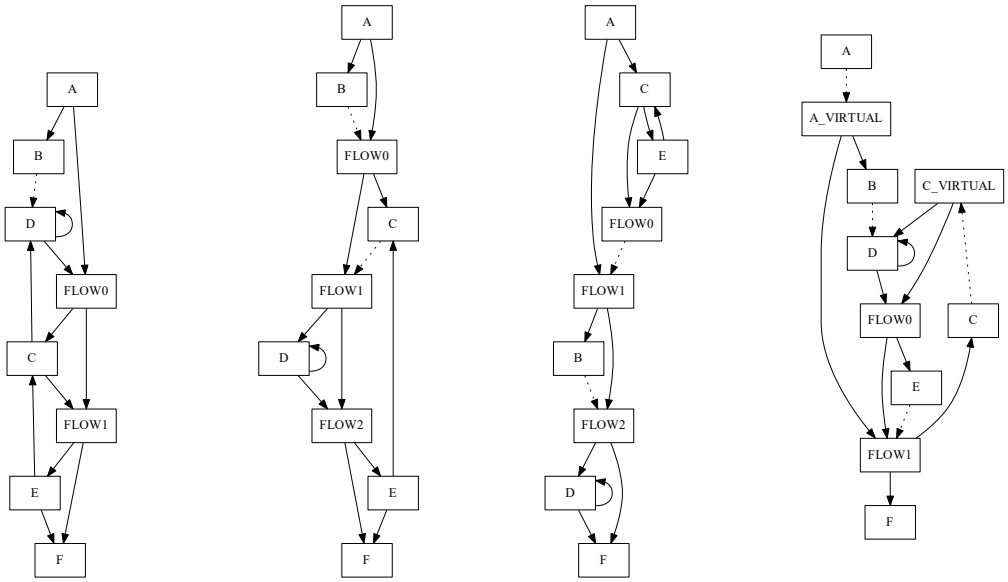
Figure 4.13 shows the outputs of four different ordering approaches for the same input CFG (Figure 4.5a). We can see that the order in which basic blocks are processed by RECONVERGE has great impact on the quality of the produced CFG. This section investigates the differences of resulting control flow of four different ordering methods:

DF Conventional depth-first traversal

DFPD Depth-first obeying post-dominance (as described by COMPUTEORDERING)

BF Conventional Breadth-first traversal

RPOT Reverse post order traversal

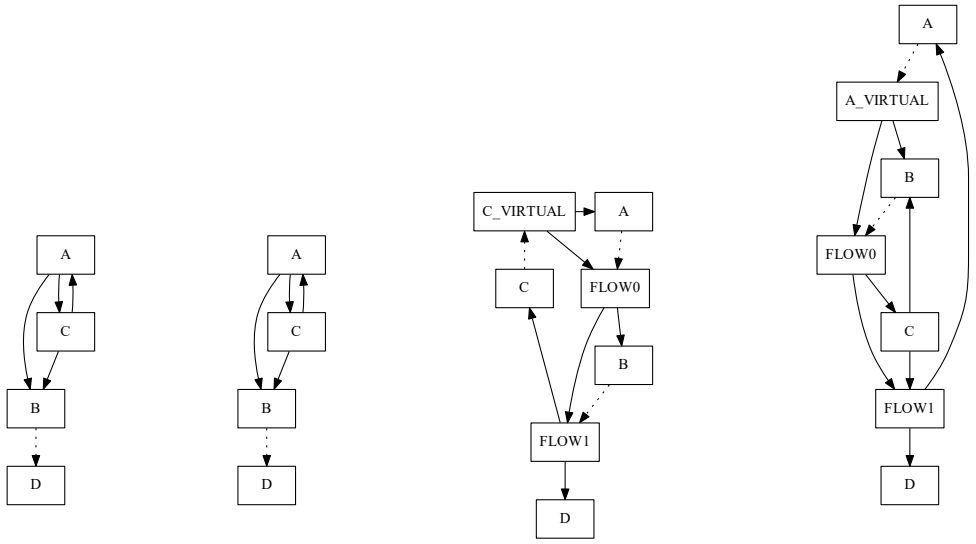


(a) DF and DFPD. (b) BF and BFPD. (c) RPOT. (d) Bottom-to-top.

Figure 4.13.: Comparison of resulting reconverging CFGs from different input orderings of input CFG shown in Figure 4.5a.

Note that for some traversal techniques (e.g. breadth-first and its post-dominance obeying variant), the resulting ordering is identical for some input CFGs and comparison of those is therefore neglected. As a convention, branch successors are iterated

in the order *true*-branch then *else*-branch when traversing the CFG and no other rule applies. This can lead to subtle differences in the output CFG. The following part focuses on orderings producing reconverging CFGs with significant features (number of flow blocks, virtual blocks, flow block ordering). The dominance region based ordering described by Moll and Hack [MH18] is neglected because only 2 out of 8 tested CFGs produced different orderings compared to **DFPD** where only one of which resulted in a different reconverged CFG (see Figure A.1).



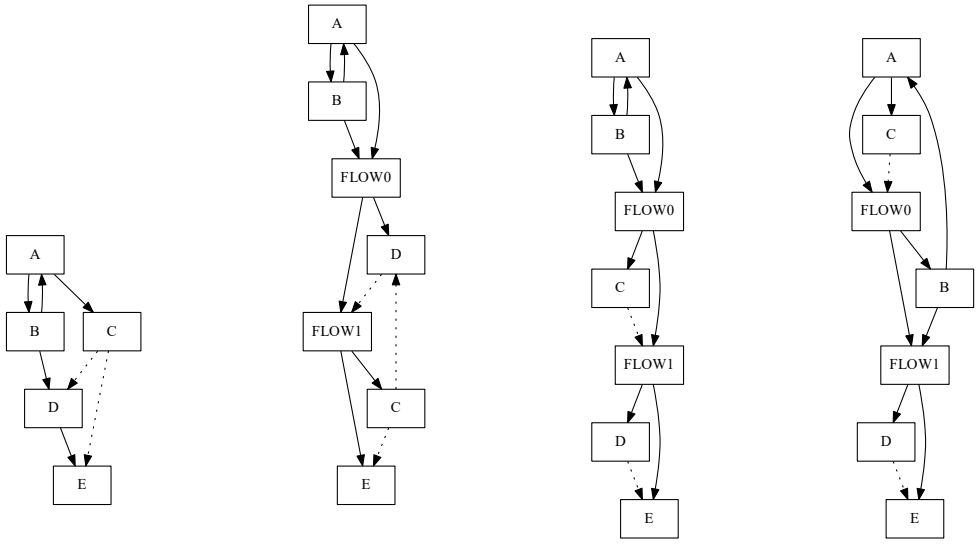
(a) Reconverging input CFG. (b) DFPD, BFPD and RPOT. (c) DF and BF. (d) Post-order: $B \to C \to A \to D$.

Figure 4.14.: Influence of ordering methods on resulting reconverging CFG. Orderings obeying dominance relations (DFPD, BFPD, RPOT) do not cause unnecessary rerouting while conventional traversals lead to insertion of *flow* and *virtual* blocks.

The first reconverging output CFG shown in Figure 4.13a is the result of a depth first traversal $A \to B \to D \to C \to E \to F$ (Sink F must come last. See original ordering of Figure 4.11a). The ComputeOrdering version of depth-first traversal (see Figure 4.11b) produces the same output CFG. Compared to the other reconverging outputs in this particular example, depth-first based traversal introduces fewest alterations by adding only 2 *flow* blocks. Breadth-first traversal caused the OT algorithm to produce 3 *flow* blocks (Figure 4.13b). Reverse post-order traversal (Figure 4.13c) is the only ordering in this example which left the backward edge $C \to E \to C$ intact by inserting the *flow*

block after C and E instead of in-between. Depending on the criteria (intact inner loop vs. number of flow blocks) RPOT might be the best choice for this particular input CFG but generalizing this result on all inputs is meaningful at this point.

Form a performance perspective, this result is significant because no additional *flow* block is placed in the inner most loop, avoiding an indirection through branching through the inserted block. To show that any ordering is feasible for correctness, an arbitrary reverse (bottom up) traversal of the CFG is tested: Ordering $D \rightarrow E \rightarrow B \rightarrow C \rightarrow A \rightarrow F$ triggers insertion of virtual nodes for A and C and the Reconverge algorithm reroutes through two additional *flow* blocks, making this result (Figure 4.13d) the most complex reconverging CFG in comparison to the first two.



(a) CFG with uniform terminator C.

(b) DF, DFPD and BFPD.

(c) BF.

(d) RPOT.

Figure 4.15.: Influence of input orderings on preserving uniform control flow when reconverging CFG of Figure (a)).

Post-dominance obeying orderings do not lead to rerouting of already reconverging CFGs. The CFG shown in Figure 4.14a remains unchanged for DFPD, BFPD and RPOT input orderings. DF, BF and post-order traversals result in virtual blocks inserted for block C and A followed by two additional *flow* blocks. Looking at the original inner loop $A \rightarrow C \rightarrow A$, both reconverging output CFGs (Figure 4.14c and 4.14d) have a minimal path length of 5 edges for the same loop: $A \rightarrow FLOW_0 \rightarrow FLOW_1 \rightarrow C \rightarrow C_{virtual} \rightarrow A$

or $A \rightarrow A_{virtual} \rightarrow FLOW_0 \rightarrow C \rightarrow FLOW_1 \rightarrow A$. This behavior clearly disqualifies conventional traversals of CFGs to generate input orderings for the reconvergence algorithm.

DF, DFPD and BFPD traversals all create the same ordering of basic blocks for the given input CFG (Figure 4.15a): $A \rightarrow B \rightarrow D \rightarrow C \rightarrow E$. Reconverging this particular ordering leads to the introduction of a new loop $D \rightarrow FLOW_1 \rightarrow C$ which did not exist in the original CFG. control flow might might pass through D twice but he resulting CFG is still semantically equivalent due to the synthesised execution mask operations. Breadth-first traversal gives a slightly different ordering ($A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$) with the same number of inserted *flow* blocks. Both reconverged CFGs of Figure 4.15b and 4.15c kept the loop $A \rightarrow B \rightarrow A$ intact because A and B were processed before C. RPOT based ordering $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E$ splits the loop by inserting the first *flow* block under condition 1 before adding B to the OT (A is *armed* predecessor and the subtree routed at A contains an open outgoing edge leading to C and not B). An important property of DFPD and BFPD orderings is preserving the original uniform control flow for edges (C, D) and (C, E).

Based on the set of tested ordering methods and input CFGs, post-dominance obeying depth-first traversal (**DFPD**) produces reconverging CFGs with few *flow* blocks, preserving uniform branches and loops in many cases, making it a suitable default choice for input orderings of the reconvergance algorithm.

5. Implementation

Before discussing the implementation details, we will first revisit the parts of a shader compilation pipeline leading from a high level language shader to the final GCN ISA output to better understand which part of the toolchain this algorithm is targeting and relate this procedure to the AMD LLVM shader compiler backend. The LLVM IR (Intermediate Representation) [LLVb] is the platform agnostic interface between the compiler frontend (like Clang or DXC) and the backend (AMDGPU in this case). LLVM IR is annotated with meta information such as calling convention, memory layout and code constraints to better reflect the properties of the target platform. However, it does not reflect wave-level control flow needed for the AMDGPU target which is why additional work needs to be done to transform the input IR (generated from a thread-level view) to GCN ISA.

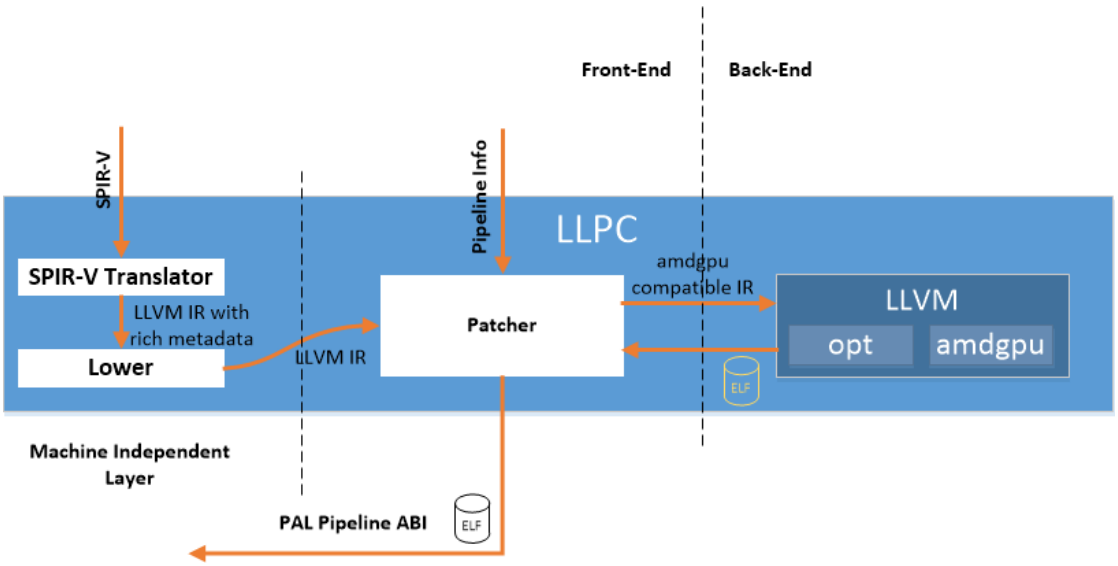


Figure 5.1.: LLPC: LLVM-Based Pipeline Compiler Architecture. Image source [Heb].

The LLPC project [Heb] is a standalone compiler toolset based on LLVM and AMD's Platform Abstraction Layer (PAL) used in the open-source Linux driver for the Vulkan® API. LLPC translates SPIR-V (a low-level IR standardized by the Khronos™ Group)

binary to LLVM IR with additional target metadata to be consumed by the LLVM optimizer (**opt**) and the **amdgpu** backend. It contains all necessary tools to compile high-level GLSL to hardware specific GCN ISA, the rough steps of the processes, including the algorithm proposed by this work are outlined below:

1. Compile GLSL or HLSL to SPIR-V using glslang or DXC
2. Translate SPIR-V to AMD flavored LLVM-IR
3. Run other LLVM optimization passes (GVN, inlining, loop optimization etc)
4. Run **ReconvergeCFG** pass
   a) Compute BB ordering from traversal of input functions CFG
   b) Prepare ordering (create nodes), add virtual nodes to reduce number of backward edges
   c) Initialize OT with virtual root node
   d) Process basic blocks in ordering
      i. Reroute under Condition 1
      ii. Add Basic Block to OT
      iii. Reroute under Condition 2
   e) Fix phi-nodes
5. Lower reconverged CFG with **SILowerReconvergingControlFlow** pass

5.1. ReconvergeCFG function pass

The code for transforming arbitrary CFGs is self-contained in a single LLVM function pass which becomes part of the scalar transformations library. The *ReconvergeCFG* pass has dependencies to the Divergence and PostDominatorTree analysis passes and requires them to be run before it is executed on the current function.

One of the requirements on the input CFG was that a unique sink exists. For graphics shaders that fact that a unique export [AMD] block must be created (by previous legalization) to output the input data for the next (possibly fixed function) pipeline stage to obey the graphics pipeline stream can be exploited at the point when the function pass is executed. Compute shaders do not have this restriction, therefore a new virtual exit block [LLVa] might be created where all previous sinks branch to.

As discussed in the algorithm chapter, nodes are closed when both *Incoming* and *Outgoing* sets are empty. *Flow* nodes contain an additional set of *FlowOutgoing* edges. When closing an edge $E = (A, B)$, successor B is removed from the *Outgoing* set of predecessor A, in the same way the A is removed from the *Incoming* set of B. If both sets of a node are empty after closing edge E, its children are moved to the parent, effectively removing the node from the OT. If as a result of closing edge E, the B's *Incoming* and *Outgoing* sets are empty, the B is closed as well. If the *Outgoing* edge of a *flow* node is closed, its target is moved to the set of *FlowOutgoing* edges. This set is later used to create the final branch instructions of *flow* node. In the case of closing (and removing) a *flow* node however, the closed outgoing edges will be used to materialize the final branch instructions of the rerouted CFG. As a result of splitting critical outgoing flow, the number of final outgoing edges of a *flow* node is either 1 or 2 depending on the existence of a remainder condition. For the first case a simple branch instruction targeting the final successor is inserted at the end of the basic block. For the second case a conditional branch instruction is created using the synthetized condition (inserted phi-instruction) for branching to the final successors.

The main part of the reconvergence algorithm is implemented in the Reroute procedure. All open outgoing edges of the set of subtrees are rerouted through a new *flow* basic block. The procedure makes a distinction between regular nodes and previously added *flow* blocks: The open outgoing edges of regular nodes are rerouted to the *flow* block directly with in-place replacements of the original branch instruction. Nodes which are *flow* nodes themselves perform the split into final successor and remaining condition outgoing flow and later materialize this newly created outgoing flow with branch instructions created upon the nodes removal from the open tree. The key mechanic here is deciding under which conditions control flow branches from the newly created *flow* basic block to its successors. This decision depends on the control flow (source basic block and its branch condition) that entered the current *flow* block. To keep track of predecessor-flow-successor conditions, a (multi-) map of source, target and condition is maintained for the course of the procedure. Finally a phi-instruction is inserted for each successor of the *flow* block, containing all conditions under which control flow from each predecessor branches to the successor.

Open *Outgoing* and *Incoming* edges of subtree nodes (with non-empty set of *Outgoing* edges) are redistributed in the following manner: the currently rerouted node is added to the *Incoming* edges of the new *flow* node and *flow* node in turn is added to the *Incoming* edges of its successors. If the current node is a flow node itself, all outgoing edges except the first closed successor (first element of *FlowOutgoing*) and their respective branch conditions are moved to the *Outgoing* set of the new *flow* node and

their predecessor-successor-condition relation is stored into the before mentioned map. The remainder condition is either the negation of first closed successors condition or a constant TRUE Boolean value if the *FlowOutgoing* set is empty. The appropriate logical NOT-instruction for the first case is inserted after the original condition is computed. The new *flow* node is added to the open outgoing edges of the current node along with previously constructed remainder condition. When the current node is a regular node and its terminator is a unconditional branch, the edge from the node to its only successor is added to the map with a constant TRUE condition. The *flow* node becomes the new target of the unconditional branch. The common pattern for a conditional branch with one *visited* successor is to replace the *unvisited* successors operand of the branch instruction with the *flow* basic block and add the original edge with a constant TRUE condition to the map. If both successors are *unvisited*, the conditional branch instruction is replaced by a unconditional branch instruction. Furthermore, the original condition and its negation (which is inserted into the nodes basic block) are added to the map along with their edges from the node to first successor and from the node to the second successor. Finally, the nodes open outgoing edges are replaced with the edges from the node to the *flow* node.

Finally after rerouting open outgoing edges, tracking their conditions and creating the phi-instructions for successors, the *flow* basic block is added to OT using AddBasicBlockToOT.

After Reconverge finished and the original CFG was changed, any preexisting phi-node in the original edge E = (A, B) that was rerouted as $E_1 = (A, X)$ and $E_1 = (X, B)$ must be adjusted to the new incoming values using LLVM's **SSAUpdater**. Therefore all predecessors of *flow* block successors are added as new incoming values to the replacement node P' (created using the SSAUpdater) of phi-nodes P found in the successor. Finally all following uses of the original phi-nodes P are replaced with updated nodes P'.

6. Conclusions

The simple definition of a *reconverging* CFG introduced in this work shows that it's powerful enough to effectively lower thread-level to wave-level control flow, solving a common problem of code generation for SPMD and SIMT applications. The underlying algorithm structurizes any input CFG, even irreducible ones, to have guaranteed points where divergent branches can re-join and preserves the original program's semantic. It retains CFGs that are already reconverging and preserves uniform control flow, both performance critical properties of the original CFG. It does so by rerouting control flow of basic blocks through new *flow* blocks if they inflict the reconvergence property. The resulting CFG now contains re-join points for divergent branches, the starting point for the lowering algorithm. The quality of the output CFG largely depends on the order in which basic blocks are processed by the algorithm. A depth-first traversal of the input CFG, obeying post-dominance relations (DFPD), appears to be one of the top candidates for the default input ordering.

The proposed algorithm is not only relevant for AMD GPUs, but for many SIMD applications with highly data-parallel workloads. No specialized predicate/reconvergence hardware stack is needed given that vector instructions can be predicated using mask operations. The LLVM based open source implementation allows developers of different platforms and architectures to leverage the simplicity of *reconverging* CFGs to better utilize the underlying hardware and improve SIMD application performance.

The Reconverge subroutine is subject to further improvements, mainly regarding the conditions under which control flow is rerouted and in which order input basic blocks are processed. Further research to simplify or even merge condition 1 and 2 might lead to easier traceability and better reasoning about suitable input orderings and resulting output control flow.

Currently the algorithm assumes all outgoing edges of *flow* blocks are divergent because the conditions of armed predecessors are divergent. Certain configurations where all inputs are uniform (e.g. *flow* blocks with unconditional branch to the next *flow* block) could be used to simplify resulting control flow, however it is unclear if such control flow can be generated by the algorithm. Further investigation in this direction might improve edge cases, but for the majority *flow* blocks will still remain divergent.

The evaluated ordering methods were judged on *overall* code quality, but it is possible that certain orderings improve a special metric such as code size or number of loops (original and artificial) which makes it possible to tune the kind of optimization for a certain target platform with different characteristics and limitations. All orderings discussed are based on the traversal of the input CFG, however it is possible that an optimal ordering cannot be found for all inputs using this approach. Control dependence graphs (CDGs) and dominator-trees might also be suitable for finding reconvergence points based on the resulting ordering. Further investigation is required to determine which properties an ordering method has to fulfill to be optimal for all input programs or at least optimal for a substantial part of the typical workloads of the target platform. If the control flow diverges only for a small subset of the typical workloads, reconverging the complete input CFG might worsen performance if for example the additional register pressure leads to spilling registers to memory. What *optimal* code is also largely depends on the target architecture. Code size for instance is more important on processors with small instruction caches, while register pressure is critical on GPUs with shared register files. A general method for divergence management that is optimal for most SIMD applications might be out of reach. Better understanding divergence in combination with other characteristics of SIMT and SPMD architectures is an important aspect for designing software and hardware solutions. A data driven solution based on optimal (possibly handcrafted) real-world shader programs could learn from the meta information and connectivity of basic blocks to generate the best ordering outside of conventional CFG traversals.

List of Figures

Bibliography

[AMD] AMD. *AMD Graphic Core Next GCN Architecture white paper*. URL: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[AR14] J. Anantpur and G. R. "Taming Control Divergence in GPUs through Control Flow Linearization." In: *Compiler Construction*. Ed. by A. Cohen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 133–153. ISBN: 978-3-642-54807-9.

[Asa+13] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover. "Convergence and Scalarization for Data-parallel Architectures." In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6494995.

[BCD12] N. Brunie, S. Collange, and G. Diamos. "Simultaneous branch and warp interweaving for sustained GPU performance." In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. May 2012, pp. 49–60. DOI: 10.1109/ISCA.2012.6237005.

[Bra+13] M. Braun, S. Buchwald, S. Hack, R. Leissa, C. Mallon, and A. Zwinkau. "Simple and Efficient Construction of Static Single Assignment Form." In: *Proceedings of the 22Nd International Conference on Compiler Construction*. CC'13. Rome, Italy: Springer-Verlag, 2013, pp. 102–122. ISBN: 978-3-642-37050-2. DOI: 10.1007/978-3-642-37051-9_6.

[CD78] N. Chapin and S. P. Denniston. "Characteristics of a Structured Program." In: *SIGPLAN Not.* 13.5 (May 1978), pp. 36–45. ISSN: 0362-1340. DOI: 10.1145/953395.953398.

[Chi17] D. Chisnall. *Modern Intermediate Representations*. May 2017. URL: https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf.

[Col] S. Collange. *GPU architecture part 2: SIMT control flow management*. URL: https://www.irisa.fr/alf/downloads/collange/cours/ada2019/ada2019_gpu_2.pdf.

[Dia+11] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. "SIMD Re-convergence at Thread Frontiers." In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: ACM, 2011, pp. 477–488. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155676.

[FLB] C. Frazier, J. Leech, and P. Brown. *The OpenGL© Graphics System: A Specification*. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.

[Fun+07] W. W. L. Fung, I. Sham, G. Yuan, and T. 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 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 407–420. ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.12.

[Haa+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the Web Up to Speed with WebAssembly." In: *SIGPLAN Not.* 52.6 (June 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363.

[Hea] J. He. *AMD Open Source Driver for Vulkan®*. URL: https://github.com/GPUOpen-Drivers/AMDVLK.

[Heb] J. He. *LLVM-Based Pipeline Compiler*. URL: https://github.com/GPUOpen-Drivers/llpc.

[Int] Intel Corporation. *Intel® SPMD Program Compiler User's Guide*. URL: https://ispc.github.io/ispc.html#experimental-support-for-ptx.

[Ken] J. Kennedy. *Use the Intel® SPMD Program Compiler for CPU Vectorization in Games*. URL: https://software.intel.com/en-us/articles/use-the-intel-spmd-program-compiler-for-cpu-vectorization-in-games.

[Knu74] D. E. Knuth. "Structured Programming with Go to Statements." In: *ACM Comput. Surv.* 6.4 (Dec. 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10.1145/356635.356640.

[Kro] T. Krogh-Jacobsen. *Unity 2018.1 release announcement, Burst Compiler (Preview)*. URL: https://blogs.unity3d.com/2018/05/02/2018-1-is-now-available/.

[Lee+14] Y. Lee, V. Grover, R. Krashinsky, M. Stephenson, S. W. Keckler, and K. Asanović. "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures." In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, United Kingdom: IEEE Computer Society, 2014, pp. 101–113. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.48.

[LG] D. Lively and H. Gruen. *Wave Programming in D3D12 and Vulkan*. URL: http://32ipi02815q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2017/07/GDC2017-Wave-Programming-D3D12-Vulkan.pdf.

[LLVa] LLVM Project. *LLVM AMDGPUUnifyDivergentExitNodes Revision 347650*. URL: https://llvm.org/svn/llvm-project/llvm/trunk/lib/Target/AMDGPU/AMDGPUUnifyDivergentExitNodes.cpp.

[LLVb] LLVM Project. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.

[LLVc] LLVM Project. *LLVM StructuizeCFG Revision 338215*. URL: https://llvm.org/svn/llvm-project/llvm/trunk/lib/Transforms/Scalar/StructurizeCFG.cpp.

[MH] M. Mantor and M. Houston. *AMD Graphic Core Next Architecture, Fusion 11 Summit presentation*. URL: http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf.

[MH18] S. Moll and S. Hack. "Partial Control-flow Linearization." In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 543–556. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192413.

[PCJ18] A. Pohl, B. Cosenza, and B. Juurlink. "Control Flow Vectorization for ARM NEON." In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*. SCOPES '18. Sankt Goar, Germany: ACM, 2018, pp. 66–75. ISBN: 978-1-4503-5780-7. DOI: 10.1145/3207719.3207721.

[PM12] M. Pharr and W. R. Mark. "ispc: A SPMD compiler for high-performance CPU programming." In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

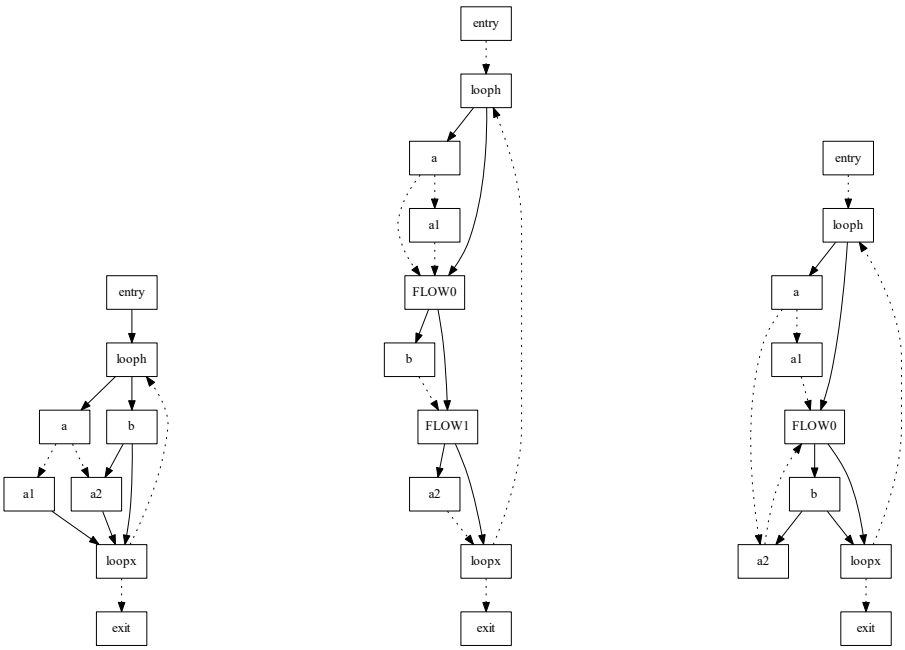
[Ren+17] B. Ren, S. Krishnamoorthy, K. Agrawal, and M. Kulkarni. "Exploiting Vector and Multicore Parallelism for Recursive, Data- and Task-Parallel Programs." In: *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '17. Austin, Texas, USA: ACM, 2017, pp. 117–130. ISBN: 978-1-4503-4493-7. DOI: 10.1145/3018743.3018763.

[Rui] M. L. Ruiz. *DirectX Intermediate Language*. URL: https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/DXIL.rst.

[SSE17] A. Sabne, P. Sakdhnagool, and R. Eigenmann. "Formalizing Structured Control Flow Graphs." In: *Languages and Compilers for Parallel Computing*. Ed. by C. Ding, J. Criswell, and P. Wu. Cham: Springer International Publishing, 2017, pp. 153–168. ISBN: 978-3-319-52709-3.

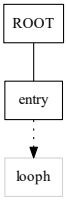
[Wil77] M. H. Williams. "Generating structured flow diagrams: the nature of unstructuredness." In: *The Computer Journal* 20.1 (1977), pp. 45–50. DOI: 10.1093/comjnl/20.1.45. eprint: /oup/backfile/content_public/journal/comjnl/20/1/10.1093/comjnl/20.1.45/2/200045.pdf.

A. Graphs

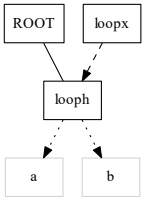


(a) Input CFG. (b) DFPD. (c) DomReg.

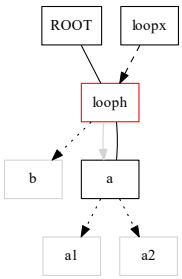
Figure A.1.: Comparison of reconverged CFGs using DFPD and dominance region (DomReg) ordering. DFPD results in an additional inserted *flow* block while DomReg introduces a backward edge.
DFPD ordering: $entry \rightarrow looph \rightarrow a \rightarrow a_1 \rightarrow b \rightarrow a_2 \rightarrow loopx \rightarrow exit$
DomReg ordering: $entry \rightarrow looph \rightarrow a \rightarrow a_1 \rightarrow a_2 \rightarrow b \rightarrow loopx \rightarrow exit$



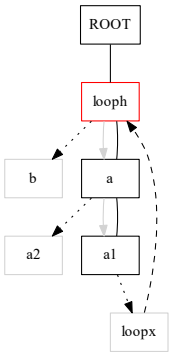
(a) Add entry.



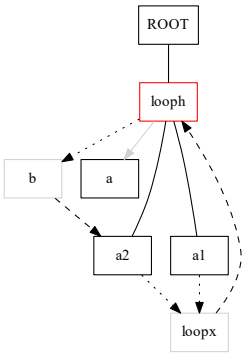
(b) Add looph.



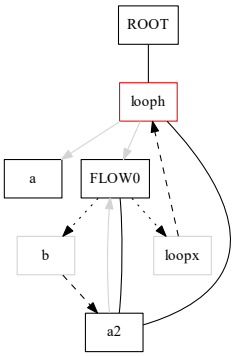
(c) Add a.



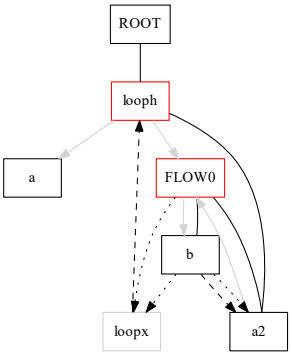
(d) Add a1.



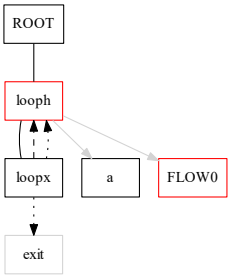
(e) Add a2.



(f) Reroute under condition 1.



(g) Add b.



(h) Add loopx.

Figure A.2.: Adding basic blocks to OT for CFG of Figure A.1a.

B. Listings

Listing B.1: Lighting computation loop corresponding to the CFG shown in Figure 1.7a. Shader code from the SteamVR runtime benchmark, Valve, 2018.

```
1  //==================== Copyright (c) Valve Corporation. All Rights Reserved. ====================
2  int nNumLightsUsed = 0;
3  int g_nNumLights = g_vNumLights.x;
4  [ loop ] for ( int i = 0; i < g_nNumLights; i++ )
5  {
6      float3 vPositionToLightRayWs = g_vLightPosition_flInvRadius[ i ].xyz - vPositionWs.xyz;
7      float flDistToLightSq = dot( vPositionToLightRayWs.xyz, vPositionToLightRayWs.xyz );
8      if ( flDistToLightSq > g_vLightFalloffParams[ i ].z ) // .z stores radius squared of light
9      {
10         continue; // Outside light range
11     }
12     if ( dot( vNormalWs.xyz, vPositionToLightRayWs.xyz ) <= 0.0 )
13     {
14         continue; // Backface cull pixel to this light
15     }
16
17     float3 vPositionToLightDirWs = normalize( vPositionToLightRayWs.xyz );
18     float flOuterConeCos = g_vSpotLightInnerOuterConeCosines[ i ].y;
19     float flTemp = dot( vPositionToLightDirWs.xyz, -g_vLightDirection[ i ].xyz ) - flOuterConeCos;
20     if ( flTemp <= 0.0 )
21     {
22         continue; // Outside spotlight cone
23     }
24     float3 vSpotAtten = saturate( flTemp * g_vSpotLightInnerOuterConeCosines[ i ].z ).xxx;
25     nNumLightsUsed++;
26
27     float flLightFalloff = DistanceFalloff( flDistToLightSq, g_vLightPosition_flInvRadius[ i ].w, g_vLightFalloffParams[ i ].xy);
28     float flShadowScalar = 1.0;
29     if ( g_vLightShadowIndex_vLightParams[ i ].x != 0 )
30     {
31         flShadowScalar = ComputeShadow_PCF_3x3_Gaussian( vPositionWs.xyz, g_matWorldToShadow[ i ] );
32         #if ( !GLSL ) // GLSL compiler doesn't like a continue inside another branch
33         {
34             if ( flShadowScalar <= 0.0 )
35                 continue;
36         }
37         #endif
38     }
39
40     #if ( GLSL )
41     {
42         if ( flShadowScalar <= 0.0 )
43             continue;
44     }
45     #endi
46
47     float2 vLightingTerms = ComputeDiffuseAndSpecularTerms(
48         g_vLightShadowIndex_vLightParams[ i ].z != 0, g_vLightShadowIndex_vLightParams[ i ].w != 0, vNormalWs.xyz,
49         vEllipseUWs.xyz, vEllipseVWs.xyz, vPositionToLightDirWs.xyz, vPositionToCameraDirWs.xyz, vDiffuseExponent.xy,
50         vSpecularExponent.xy, vSpecularScale.xy, flReflectance, flFresnelExponent );
51
52     float3 vLightColor = g_vLightColor[ i ].rgb;
53     float3 vLightMask = vLightColor.rgb * flShadowScalar * flLightFalloff * vSpotAtten.rgb;
54     o.vDiffuse.rgb += vLightingTerms.xxx * vLightMask.rgb;
55     o.vSpecular.rgb += vLightingTerms.yyy * vLightMask.rgb;
56 }
```