TAMING IRREGULAR CONTROL-FLOW WITH TARGETED COMPILER TRANSFORMATIONS

by

Charitha Saumya Gusthinna Waduge

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Milind Kulkarni, Chair

School of Electrical and Computer Engineering

Dr. Samuel P. Midkiff

School of Electrical and Computer Engineering

Dr. Timothy Rogers

School of Electrical and Computer Engineering

Dr. Xiaokang Qiu

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

To my parents, for their unconditional love, support, and encouragement.

ACKNOWLEDGMENTS

Deciding to pursue a PhD in the United States was one of the toughest decisions I had to make in my life. I knew that being away from my family and friends back home would be an unbearable challenge. Looking back, I think I made the best decision of my life. The experiences I had and the great people I have met along the way have shaped me into a better person. I would like to thank all those people who have helped me in this journey.

First, I would like to thank my brilliant advisor, Dr. Milind Kulkarni for his excellent mentorship and guidance. I greatly admire his kindness, humility and ability to motivate and inspire his students. I am fortunate to have him as my advisor. I would like to thank Dr. Timothy Rogers for providing me access to GPU hardware in his lab which greatly helped me to advance my research on GPU code optimizations. I express my sincere gratitude to Dr. Samuel Midkiff, Dr. Timothy Rogers and, Dr. Xiaokang Qiu for serving in my PhD advisory committee. Their valuable insights and feedback have been crucial in shaping my research.

My labmate and friend, Kirshanthan Sundarajah has been an excellent collaborator in my research. I like to extend my thanks to him for our lengthy and fruitful research discussions and his various contributions in fine-tuning up my dissertation research. I would like to thank my external collaborators Dr. Pavlos Petoumenos and Dr. Rodrigo Rocha for their help and feedback on my research about CPU code size optimizations. It would not be complete without extending my gratitude to all my PLCL (Parallelism, Languages, and Compilers Lab) labmates. All the fun discussions and exciting outings gave me the much-needed distractions from research to relax myself.

I extend my sincere gratitude to my parents, my in-laws and my sister for their constant love, support and encouragement. Finally, I express my heartfelt gratitude to my loving wife Rangana. She has been a friend, loving critic and a great source of motivation. Without her love and support I could not have finished this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Irregular control-flow structures like deeply nested conditional branches are common in real-world software applications. Improving the performance and efficiency of such programs is often challenging because it is difficult to analyze and optimize programs with irregular control flow. We observe that real-world programs contain similar or identical computations within different code paths of the conditional branches. Compilers can merge similar code to improve performance or code size. However, existing compiler optimizations like code hoisting/sinking, and tail merging do not fully exploit this opportunity. We propose a new technique called Control-Flow Melding (CFM) that can merge similar code sequences at the control-flow region level. We evaluate CFM in two applications. First, we show that CFM reduces the control divergence in GPU programs and improves the performance. Second, we apply CFM to CPU programs and show its effectiveness in reducing code size without sacrificing performance. In the next part of this dissertation, we investigate how CFM can be extended to improve dynamic test generation techniques like Dynamic Symbolic Execution (DSE). DSE suffers from path explosion problem when many conditional branches are present in the program. We propose a non-semantics-preserving branch elimination transformation called CFM-SE that reduces the number of symbolic branches in a program. We also provide a framework for detecting and reasoning about false positive bugs that might be added to the program by non-semantics-preserving transformations like CFM-SE. Furthermore, we evaluate CFM-SE on real-world applications and show its effectiveness in improving DSE performance and code coverage.

1. INTRODUCTION

Control-flow irregularity and its impact on performance and efficiency of programs is a well-studied problem in programming languages and compilers research community [1]–[3]. In programs with *regular* control-flow the program execution path and memory access patterns exercised by the program are predictable. In other words, in regular programs program behavior does not change much when the input data changes. Classical examples of such programs are matrix multiplications and tensor contractions. Therefore, reasoning about the behavior of regular programs and optimizing them is less cumbersome compared to irregular programs. In contrast, the run-time characteristics of *irregular* programs is highly dependent on the input data or properties of the execution environment (such as the thread identifier in GPU programs). This makes it challenging to analyze, optimize and verify irregular programs. Irregular control-flow realize in programs with deeply nested conditional branches (*i.e.* `if-then`, `if-then-else`, `switch-case` nested inside each other). In this work, we focus on how to improve the performance, efficiency, and testability of programs with irregular control-flow.

The performance of GPU programs is sensitive to the control-flow structure of the program. GPU programs with data dependent control-flow (*i.e.* divergent branches) are often impacted by control-flow divergence [4]. GPU programming model allows Single-Program-Multiple-Data (SPMD) programming style where the programmer has to write a single program that can be executed by thousands of threads in parallel. This hides away the complexity of the hardware from the programmer. Behind the scenes, the GPU hardware executes groups the threads into *warps* or *wavefronts*, and each thread group execute the instructions of the program in lock-step. This is called Single-Instruction-Multiple-Data (SIMD) execution model. SIMD execution model is crucial for achieving high efficiency of GPU execution. However, divergent branches can cause performance bottlenecks in hardware that employs SIMD-style execution. This is because, the GPU hardware has to serialize the execution of the threads in a warp if different threads want to take different control-flow paths at divergent branches. If the GPU program contains many input-dependent or thread-ID dependent divergent branches (*i.e.* irregular control-flow), this serialization can result in

a significant performance degradation. Many architectural modifications [5]–[8] have been proposed to reduce the impact of control-flow divergence in GPUs. Even though highly effective, architectural modifications are often expensive and not always feasible. Software only techniques [9]–[12] have been investigates as well to reduce the impact of control-flow divergence by using various program transformations. One of the key compiler technique for reducing control-flow divergence it to merge (or fuse) similar code within divergent branches. Techniques like branch fusion [11] and tail merging [13] are based on this idea. However, these techniques are merging instructions at the basic block level and, they are not general enough to merge complex control-flow regions.

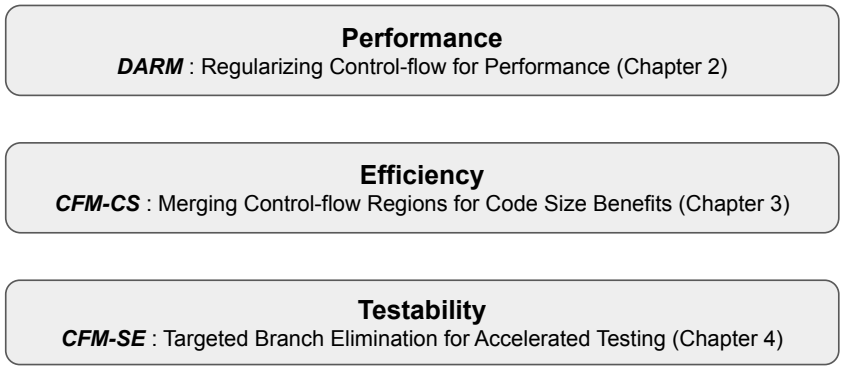

Figure 1.1. Techniques proposed for mitigating the impact of irregular control-flow on program Performance, Efficiency, and Testability

In resource constrained environments like embedded systems, size of the executable program becomes a critical optimization goal. In modern compilers like LLVM [14] and GCC [15], there are dedicated optimizations and optimization levels (such as `-Oz`) to reduce the size of the executable program. Well-known code size reduction optimizations include dead code elimination [16], common subexpression elimination [17], and procedural abstraction [13], [18]. Programs with irregular control-flow also presents us an important opportunity for code size reduction. In programs with many conditional branches, the divergent code paths of each branch often times contains similar or identical code. These similar code sequences can be merged to reduce the size of the program. Optimizations like code hoisting/sinking and tail merging [13] uses this observation to reduce code size. Despite

these advancements, there are still many missed opportunities in real-world applications to reduce the size of the executable program. In particular, existing compiler techniques only exploit code similarity in very restricted scenarios. Code sinking only works when a set of basic blocks have identical code at the end and, they have a common successor. Similarly, code hoisting only works when a set of basic blocks have identical code at the beginning and, they have a common predecessor. Generalized tail merging is capable of merging structurally similar control-flow regions, but the matching basic blocks must contain identical or nearly identical code. There is a need for a general compiler transformation that can exploit both control-flow region level instruction level code similarity to reduce the size of programs.

Irregular control-flow structures pose a significant challenge in testing and verification of programs. The number of feasible program paths in a program grows exponentially with the number of input-dependent conditional branches present in a program. Path enumeration based test generation techniques like Dynamic Symbolic Execution [19] and Concolic Execution [20] are not scalable for programs with many conditional branches. This is the well-known the *Path Explosion Problem* in dynamic test generation [21]–[23]. Existing solutions proposed for path explosion problem include guiding symbolic execution towards the most interesting program paths by using heuristics [24], [25], merging similar program states to reduce the number of explored paths [26], producing summaries of program behavior to avoid exploring similar paths [27], [28], or isolating code regions from its surrounding environment to reduce the state explosion [29]. Another important but less-studied class of techniques is applying program transformations in a targeted fashion to reduce the overheads of symbolic execution [30]–[32]. In fact, Collingbourne et al. [33] have shown that applying compiler transformations to remove conditional branches can significantly reduce the path exploration overheads in symbolic execution. Control-flow irregularity presents us an opportunity to apply such compiler transformations to improve the testability of those programs. However, existing compiler transformations are designed to improve the performance and therefore, they are ill-suited for improving the testability.

In this dissertation, we make the following contributions to address the above shortcomings of existing compiler techniques:

- We developed DARM, a compiler technique for reducing control-flow divergence in GPU programs. DARM is based on the observation the divergent code paths in GPU programs often contain similar code sequences. DARM detects this code similarities using statically analyzing the program and, decides whether it is profitable to merge these divergent code paths. If proven profitable according to a cost model, DARM applies a novel transformation called *Control-flow Melding* to merge instructions that are otherwise executed divergently. DARM is capable of exploiting both basic block level and control-flow regions level code similarity and, therefore more general than existing compiler techniques. We implement DARM in LLVM and show that it can improve the performance of several important GPU benchmarks on AMD GPUs.

- We developed CFM-CS, an application of control-flow melding to reduce code size in CPU programs. CFM-CS implementation is more general than DARM and, it can merge more complex control-flow regions that are present in real-world CPU programs. Our evaluation of CFM-CS shows that it can be useful in reducing code size in numerous large real-world benchmarks with minimal runtime and compile-time overheads.

- We developed CFM-SE, a targeted non-semantics-preserving control-flow transformation designed to improve the scalability of dynamic symbolic execution (DSE) of a given program. First CFM-SE uses static analysis to identify expensive symbolic conditional branches for DSE to explore. Then, CFM-SE inserts a minimal number of *dead* instructions to the basic blocks of these branches to make them have identical computation sequences. Finally, CFM-SE uses control-flow melding to merge these basic blocks into a single basic block *without* inserting any additional branches. By branch elimination, CFM-SE statically merges program states and drastically reduce the number of solver queries required by DSE. We also built a framework for detecting any false positive crashes that might be introduced by CFM-SE due to its non-semantics-preserving nature. Our evaluation shows that CFM-SE can reduce the number of solver queries significantly, accelerate the bounded verification of programs and, improve the coverage on real-world programs compared naive DSE and dynamic state merging.

2. DARM: CONTROL-FLOW MELDING FOR SIMT THREAD DIVERGENCE REDUCTION

2.1 Introduction

General Purpose Graphics Processing Units (GPGPU) are capable of executing thousands of threads in parallel, efficiently. Advancements in the programming models and compilers for GPUs have made it much easier to write data-parallel applications. Unfortunately, exploiting data parallelism does not immediately translate to better performance. One key reason for the lack of performance portability is that GPGPUs are not capable of executing all the threads independently. Instead threads are grouped together into units called *warps*, and threads in a warp execute instructions in lockstep. This is commonly referred to as the Single Instruction Multiple Thread (SIMT) execution model.

The SIMT model suffers performance degradation when threads exhibit *irregularity* and can no longer execute in lockstep. Irregularity comes in two forms, irregularity in memory accesses patterns (*i.e.* memory divergence) and irregularity in the control-flow of the program (*i.e.* control-flow divergence). Memory divergence occurs when GPGPU threads needs to access memory at non-uniform locations, which results in un-coalesced memory accesses. Un-coalesced memory accesses are bad for GPU performance because memory bandwidth can not be fully utilized to do useful work.

Control-flow divergence occurs when threads in a warp diverge at branch instructions. At the *diverging* branch, lockstep execution can not be maintained because threads in a warp may want to execute different basic bocks (*i.e.* diverge). Instead, when executing instructions along a diverged path, GPGPUs mask out the threads that do not want to take that path. The threads *reconverge* at the Immediate Post-DOMinator (IPDOM) of a divergent branch—the instruction that all threads from both branches want to execute. This style of IPDOM-based reconvergence is implemented in hardware in most GPGPU architectures to maintain SIMT execution. Even though IPDOM-based reconvergence can handle arbitrary control-flow, it imposes a significant performance penalty if a program has a lot of divergent branches. In the IPDOM reconvergence model, instructions executed on divergent branches necessarily cannot utilize the full width of a SIMD unit. If the code has

a lot of nested divergent branches or divergent branches inside loops, this style of execution causes significant under-utilization of SIMD resources.

For some GPGPU applications divergent branches are unavoidable, and there have been many techniques proposed to address this issue both in hardware and software. Proposals such as Dynamic warp formation [34], Thread block compaction [6] and Dual-path execution [5] focus on mitigating the problem at the hardware level by changing how threads are scheduled for execution and making sure that threads following the same path are grouped together. Unfortunately, such approaches are not useful on commodity GPGPUs.

Table 2.1. Comparison of techniques for divergence reduction

Control-flow and Instruction Pattern	**Technique**		
	Tail Merging	Branch Fusion	DARM
Diamond control-flow with identical instruction sequences	✓	✓	✓
Diamond control-flow with distinct instruction sequences	✗	✓	✓
Complex control-flow	✗	✗	✓

There have also been efforts to reduce divergence through compiler approaches that leverage the observation that different control-flow paths often contain similar instruction (sub)sequences. *Tail merging* [13] identifies branches that have identical sequences of code and introduces early jumps to *merged* basic blocks, with the effect of reducing divergence. *Branch fusion* generalizes tail merging to work with instruction sequences that may not be identical [11]. However, branch fusion cannot analyze complex control-flow and hence it is restricted to simple *if-then-else* branches where each path has a single basic block (*i.e.* diamond-shaped control-flow).

In this work, we propose a more general, software-only approach of exploiting similarity in divergent paths, called *control-flow melding*. Control-flow melding is a general control-flow transformation which can meld similar control-flow *subgraphs* inside a *if-then-else* region (not just individual basic blocks). By working hierarchically, recursively melding divergent control-flow at the level of subgraphs of the CFG, control-flow melding can handle substantially more general control structures than prior work. We describe DARM, a realization of

control-flow melding for general GPGPU programs. Table 2.1 compares the capabilities of DARM with branch fusion and tail merging.

DARM works in several steps. First, it detects divergent *if-then-else* regions and splits the divergent regions into Single Entry Single Exit (SESE) control-flow subgraphs. Next it uses a hierarchical sequence alignment technique to *meld* profitable control-flow subgraphs, repeatedly finding subgraphs whose control-flow structures and constituent instructions can be aligned. Once a fixpoint is reached, DARM uses this hierarchical alignment to generate code for the region with reduced control-flow divergence.

The main contributions of this work are,

- **D**ivergence-**A**ware-**R**egion-**M**elder (DARM), a realization of control-flow melding that identifies profitable melding opportunities in divergent *if-then-else* regions of the control-flow using a hierarchical sequence alignment approach and then melds these regions to reduce control-flow divergence.

- An implementation of DARM in LLVM [14] that can be applied to GPGPU programs written in HIP [35] or CUDA [36]. Our implementation of DARM is publicly available as an archival repository[1] and up-to-date version is available in GitHub[2].

- An evaluation of DARM on a set of synthetic GPU programs and a set of real-world GPU applications showing its effectiveness

2.2 Background

2.2.1 GPGPU Architecture and Programming Models

Modern GPGPUs are designed to have much higher instruction throughput and memory bandwidth compared to CPUs while still being within a reasonable cost and power consumption budget. The specific design of GPGPUs makes them well-suited for highly data parallel applications like scientific simulations, computer graphics and machine learning. GPGPUs have multiple processing core clusters called *streaming multiprocessors (NVIDIA)* or *compute units (AMD)*. Each of these clusters contain multiple parallel lanes (*i.e.* SIMT cores), a

[1] ↑https://doi.org/10.5281/zenodo.5784768
[2] ↑https://github.com/charitha22/cgo22ae-darm-code

vector register file, L1 cache and a chunk of shared memory. The unit of execution is called a warp (or wavefront). A warp is a collection of threads executed in lock-step on a SIMT core. Shared memory is shared among the warps executing on a core cluster. All the core clusters can access unified L2 cache and global memory. Global memory is accessed via a high bandwidth interconnection network. A branch unit takes care of control-flow divergence by maintaining a SIMT stack to enforce IPDOM based reconvergence, as discussed in Chapter 2.1. A single core cluster is capable of executing thousands of threads in parallel. Whenever a warp hits a long latency memory instruction the *warp scheduler* schedules a different warp on that SIMT core. This allows GPGPUs to hide the latency of expensive memory reads and writes.

GPGPU programming abstractions like CUDA [36] or HIP [35] gives the illusion of data parallelism with independent threads. Programmer only has to worry about what a single thread supposed to do and specify that using the programming model. This is also referred to as Single-Program-Multiple-Data (SPMD) programming model [37] because multiple instances of the same program are executed by many threads independently with each thread operating on different data elements. These threads are organized in a hierarchy of up to 3 dimensional *blocks* and *grids*. This makes it easier to map computations over 1, 2 or 3-dimensional spaces into different threads. Programming model also provides shared memory synchronization primitives to communicate between threads in a thread block and warp level primitives [38] to communicate between threads in a warp. During real execution on the GPGPU hardware, a group of program instances (*i.e.* threads) are mapped to a warp and executed in lock-step. Therefore, threads taking vastly different control-flow paths during real execution–due to control-flow divergence–is highly detrimental to the performance because of the SIMT execution limitations.

2.2.2 LLVM SSA form and GPU Divergence Analysis

LLVM [14] is a general framework for building compilers, optimizations and code generators. Most of the widely adopted GPGPU compilers [39], [40] are built on top of the LLVM infrastructure. LLVM uses a target-independent intermediate representation, LLVM-

IR, that enables implementing portable compiler optimizations. LLVM-IR uses static single assignment form [41] which requires that every program variable is assigned once and is defined before being used. SSA form uses ϕ nodes to resolve data-flow when branches are present, selecting which definition should be chosen at a confluence of different paths.

In GPGPU compilers, a key step in identifying divergent control-flow regions is performing compiler analyses to identify divergent variables (or branches) [11], [42]. A branch is divergent if the branching condition evaluates to a non-uniform value for different threads in a warp. If the branching condition is divergent, threads in a warp will have to take different control-flow paths at this point. Impact of control-flow divergence has extensively studied in different contexts [43]–[46]. Reducing control-flow divergence requires finding the source of divergence in a program. Structured CFGs make it easier to produce a precise divergence analysis. Sabne *et al.* has provided a new formalization for structured CFGs and showed the importance of structured CFGs in thread divergence [47]. LLVM's divergence analysis tags a branch as divergent, if the branching condition is either data-dependent or sync-dependent on a divergent variable (such as thread ID) [42], [48]. If some variable `x` is data-dependent on a divergent variable `y`, then `x` also becomes divergent. This is called data-dependent divergence. On the other hand, sync-dependancy characterizes the control-flow aspect of divergence. For example, consider a *ph*i node at a confluence point of `if` and `then` control-flow paths and assume that the branching condition is dependent on thread ID. Now the *ph*i node also becomes divergent even if does not have a direct data dependence with a divergent variable. This is because the resolved value of the *ph*i node during execution depends on which control-flow path is taken to reach it (`if` or `then` path). In this case the *ph*i node is sync-dependent on the thread ID. LLVM's divergence analysis keeps track of these data-dependent and sync-dependent divergent variables and provide an interface to query if a given variable is divergent or not. Coutinho *et al.* constructed a divergence analysis to statically identify variables with same value for every thread executed on a SIMT core and used this analysis to drive *Branch Fusion* [11]. Divergence analysis is often conservative meaning that it will often classify a variable as divergent even though it is not in practice. To reduce the number of false positive in divergence analysis, authors proposed a method to split the live ranges of certain loop variables. Reducing the live range in those variables

enable the analysis to correctly classify them as being non-divergent avoiding the number of false positives. Recently, Rosemann *et al.* has presented a more general and precise divergence analysis for reducible CFGs based on abstract interpretation for uniformity [49]. This version of divergence analysis is adapted in more recent versions of LLVM due to its generality and speed. Having access to precise divergence analysis aids the compiler to correctly identify divergent branches in the CFG and apply control-flow optimizations such as tail merging and branch fusion to reduce control-flow divergence.

2.3 Overview of DARM

```
1  __global__ static void bitonicSort(int *values) {
2    // copy data from global memory to shared memory
3    __syncthreads();
4    for (unsigned int k = 2; k <= NUM; k *= 2) {
5      for (unsigned int j = k / 2; j > 0; j /= 2) {
6        unsigned int ixj = tid ^ j;
7        if (ixj > tid) {
8          if ((tid & k) == 0) {
9            if (shared[ixj] < shared[tid])
10             swap(shared[tid], shared[ixj]);
11         }
12         else {
13           if ( shared[ixj] > shared[tid])
14             swap(shared[tid], shared[ixj]);
15         }
16       }
17       __syncthreads();
18     }
19   } // write data back to global memory
20 }
```

Figure 2.1. Bitonic sort kernel

We use *Bitonic Sort* kernel as a running example to explain how DARM can be used to reduce control-flow divergence in programs with complex control-flow. Bitonic Sort is a kernel used in many parallel sorting algorithms such as bitonic merge sort and Cederman's quicksort [50], [51]. Figure 2.1 shows a CUDA implementation of bitonic sort. This kernel is our running example for describing DARM's control-flow melding algorithm.

In this kernel, the branch condition at line 8 depends on the *thread ID*. Therefore, it is divergent. Threads arriving at this branch may want to take the *true* or *false* paths depending on the outcome of the branch condition resulting in control-flow divergence. Since the divergent branch is located inside a loop, the execution of the two sides of the branch needs to be serialized many times, resulting in high control-flow divergence. However the code inside the `if` (line 9-10) and `else` (line 13-14) sections of the divergent branch are similar in two ways. First, both code sections have the same control-flow structure (*i.e. if-then* branch). Second, instructions along the two paths are also similar. Both conditions compare two elements in the `shared` array and perform a `swap` operation. Therefore the contents of the `if` and `else` sections can be melded to reduce control-flow divergence. Both code sections consists of shared memory loads and store operations. In the unmelded version of the code these shared memory operations will have to be serialized due to thread-divergence. However, if the two sections are melded threads can issue the memory instructions in the same cycle resulting in improved performance.

Existing compiler optimizations such as tail merging and branch fusion cannot be applied to this case. Tail merging is applicable only if two basic blocks have a common destination and have identical instruction sequences at their tails. However in bitonic sort, the `if` and `else` sections of the divergent branch have multiple basic blocks, and the compiler cannot apply tail merging. Similarly branch fusion requires diamond shaped control-flow and does not work if the `if` and `else` sections of the branch contain complex control-flow structures.

DARM solves this problem in two phases. In the analysis phase (Section 2.4.3), DARM analyzes the control-flow region dominated by a divergent branch to find isomorphic sub-regions that are in the true and false paths of the divergent branch. These isomorphic sub-region pairs are aligned based on their melding profitability using a sequence alignment strategy. Melding profitability is a compile-time approximation of the percentage of thread cycles that can be saved by melding two control-flow regions. Next, DARM choses profitable sub-region pairs in the alignment (using a threshold) and computes an instruction alignment for corresponding basic blocks in the two regions. In the code generation phase (Section 2.4.4), DARM uses this instruction alignment to meld corresponding basic blocks in the sub-region pair. This melding is applied iteratively until no further profitable melding can be performed.

DARM's melding transformation is done in SSA form, therefore the resulting CFG can be optimized further using other compiler optimizations (Sections 2.4.5 and 2.4.6).

2.4 Detailed Design

In this section we describe the algorithm used by DARM to meld similar control-flow subgraphs. First we define the following terms used in our algorithm description.

2.4.1 Preliminaries and Definitions

Definition 2.4.1. ***Simple Region*** *: A simple region is a subgraph of a program's CFG that is connected to the remaining CFG with only two edges, an entry edge and an exit edge.*

Definition 2.4.2. ***Region*** *: A region of the CFG is characterized by two basic blocks, its entry and exit. All the basic blocks inside a region are dominated by its entry and post-dominated by its exit. Region with entry E and exit X is denoted by the tuple (E, X). LLVM regions are defined similarly [52], [53].*

Definition 2.4.3. ***Single Entry Single Exit Subgraph*** *: Single entry single exit (SESE) subgraph is either a simple region or a single basic block with a single predecessor and a successor.*

Note that a region with entry E and exit X can be transformed into a simple region by introducing a new entry and exit blocks E_{new}, X_{new}. All successors of E are moved to E_{new} and E_{new} is made the single successor of E. Similarly, all predecessors of X are moved to X_{new} and a single exit edge is added from X_{new} to X.

Definition 2.4.4. ***Simplified Region*** *: A region with all its subregions transformed into simple regions is called a simplified region.*

We now turn to the steps the DARM compiler pass takes to reduce control divergent code.

2.4.2 Detecting Meldable Divergent Regions

First DARM needs to detect divergent branches in the CFG. We use LLVM's built-in divergence analysis to decide if a branch is divergent or not (Section 2.2). The smallest

CFG region enclosing a divergent branch is called the *divergent region* corresponding to this branch. Melding transformation is applied only to divergent regions of the CFG. The next step is to decide if a divergent region contains control-flow subgraphs (definition 2.4.3) that can be safely melded.

Definition 2.4.5. ***Meldable Divergent Region****: A simplified region R with entry E and exit X is said to be meldable and divergent if the following conditions are met,*

1. *The entry block of R has a divergent branch*
2. *Let B_T and B_F be the successor blocks of E. B_T does not post-dominate B_F and B_F does not post-dominate B_T*

According to definition 2.4.5, a *meldable divergent region* has a divergent branch at its entry (condition 1). This makes sure that our melding transformation is only applied to divergent regions, and non-divergent parts of the control-flow are left untouched. Condition 2 ensures that paths $B_T \rightarrow X$ (*i.e.* true path) and $B_F \rightarrow X$ (*i.e.* false path) consists of at least one SESE subgraph and these subgraphs from the two paths can potentially be melded to reduce control-flow divergence. Consider our running example in Figure 2.1. When this kernel is compiled with *ROCm HIPCC* GPU compiler [35] with `-O3` optimization level into LLVM-IR, we get the CFG shown in Figure 2.5a. Note that the compiler aggressively unrolls both the loops (lines 4 and 5) in the kernel, and the resulting CFG consists of multiple repeated segments of the inner loop's body (lines 6-17). In Figure 2.5a, only one unrolled instantiation of the loop body is shown. As explained in Section 2.3, this kernel contains a divergent branch, which is at the end of basic block $\%B$. Also $\%B$'s two successors $\%C$ and $\%D$ do not post-dominate each other. Therefore the region $(\%B, \%G)$ is a meldable divergent region.

2.4.3 Computing Melding Profitability

Definition 2.4.5 only allows us to detect regions that may contain meldable control-flow subgraphs. It does not tell us whether it is legal to meld them or melding them will improve

performance. First we need to define what conditions needs to be satisfied for two SESE subgraphs to be meldable.

Definition 2.4.6. ***Meldable SESE Subgraphs****: SESE subgraphs $S1$ and $S2$ where $S1$ belongs to the true path and $S2$ belongs to the false path are meldable if any one of the following conditions are satisfied,*

1. *Both $S1$ and $S2$ have more than one basic block and they are structurally similar i.e. isomorphic.*
2. *$S1$ is a simple region and $S2$ consists of a single basic block or vice versa.*
3. *Both $S1$ and $S2$ consists of single basic block.*

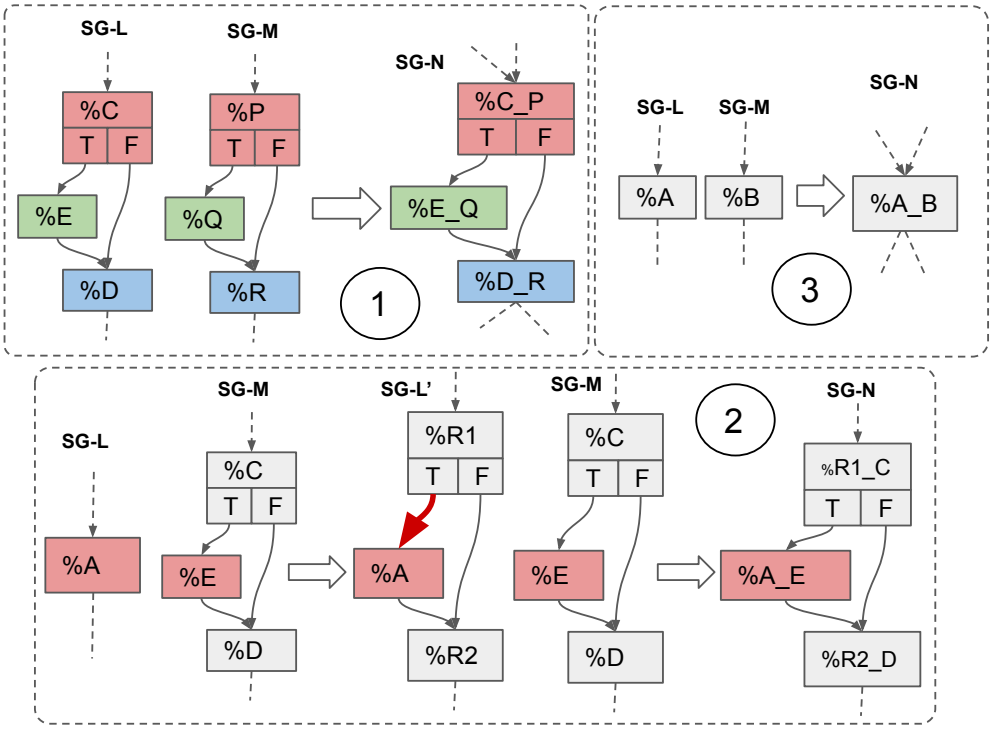

Figure 2.2. Examples showing the 3 cases considered by DARM to detect meldable subgraphs

Definition 2.4.6 ensures that any two SESE subgraphs that meets any one of these conditions can be melded without introducing additional divergence to the control-flow. Note that we do not consider subgraphs that contain *warp-level intrinsics*[38] for melding because melding

such subgraphs can cause deadlock. Figure 2.2 shows three examples where each of the above conditions are applicable. Assume in each example subgraphs L and M are in a divergent region (E, X) and only one of the subgraphs are executed from any program path from E to X. (*i.e.* any thread in warp that executes E must either go through L or M but not both).

Region-Region Melding : In case ①, two SESE subgraphs L and M are isomorphic, therefore they can be melded to have the same control-flow structure (subgraph N in Figure 2.2-①). In the melded subgraph N, basic blocks $\%C\_P$ and $\%D\_R$ are guaranteed to post-dominate E and threads can reconverge at these points resulting in reduction in control-flow divergence. Also the structural similarity in case ① ensures that we do not introduce any additional branches into the melded subgraph.

Basic block-Region Melding : In case ②, basic block $\%A$ (in subgraph L) can potentially be melded with any basic block in CFG M. Assume that basic blocks $\%A$ and $\%E$ have the most melding profitability (melding profitability described later). First we replicate the control-flow structure of M to create a new CFG L'. Then we place $\%A$ in L' such that $\%A$ and $\%E$ are in similar positions in the the two CFGs L' and M. We also ensure the correctness of the program by concretizing the branch conditions in L' to always execute $\%A$ and create ϕ nodes at dominance frontiers of $\%A$ to make sure values defined inside $\%A$ are reached to their users [41]. In this example branch at end of basic block $\%R1$ will always take the edge $\%R1 - \%A$ (bold arrow in subgraph L') and ϕ nodes will be added to $\%R2$. Now subgraphs L' and M are isomorphic and therefore can be melded similar to case ①. We refer to this process as *Region Replication*. Main benefit of region replication is that it allows us to meld $\%A$ with any profitable basic block in subgraph M and resultant subgraph N has less divergence because threads can reconverge at basic blocks $\%R1\_C$ and $\%R2\_D$ in melded subgraph N.

Basic block-Basic block Melding : Case ③ is the simplest form where two SESE basic blocks are melded.

A meldable divergent region can potentially have multiple SESE subgraphs in its true and false paths. Therefore we need a strategy to figure out which subgraph pairs to meld. We formulate this as a sequence alignment problem as follows. First, we obtain a ordered sequence of subgraphs in true path and false of the divergent region. Subgraphs are ordered

using the post-dominance relation of their entry and exit blocks. For example, if entry node of subgraph S_2 post-dominates exit node of subgraph $S1$, then $S2$ comes after $S1$ in the order and denoted as $S1 \prec S2$. A subgraph alignment is defined as follows,

Definition 2.4.7. ***Subgraph Alignment****: Assume a divergent region (E, X) has ordered SESE subgraphs $\{S_1^T, S_2^T, \ldots, S_m^T\}$ in its true path and ordered subgraphs $\{S_1^F, S_2^F, \ldots, S_n^F\}$ in the false path. A subgraph alignment is an ordered sequence of tuples,*
$A = \{(S_{\mathrm{i}0}^T, S_{\mathrm{j}0}^F), (S_{\mathrm{i}1}^T, S_{\mathrm{j}1}^F), \ldots, (S_{\mathrm{i}k}^T, S_{\mathrm{j}k}^F)\}$
where,

1. *if $(S_p^T, S_q^F) \in A$ then S_p^T and S_q^F are meldable subgraphs*
2. *if $(S_{p1}^T, S_{q1}^F) \prec (S_{p2}^T, S_{q2}^F)$ then $S_{p1}^T \prec S_{p2}^T$ and $S_{q1}^T \prec S_{q2}^T$*

According to definition 2.4.7, only meldable subgraphs are allowed in a alignment tuple and if the aligned subgraphs are melded, the resultant control-flow graph does not break the original dominance and post-dominance relations of the subgraphs.

Given a suitable alignment scoring function F and gap penalty function W, we can find an optimal subgraph alignment using a sequence alignment method such as Smith-Waterman [54] algorithm. The scoring function F measures the profitability of melding two meldable subgraphs $S1$ and $S2$. Prior techniques have employed instruction frequency to approximate the profit of merging two functions [55], [56]. We use a similar method to define subgraph melding profitability. First we define the melding profitability of two basic blocks $b1$ and $b2$ as follows,

$$MP_B(b1, b2) = \frac{\sum_{\mathrm{i} \in Q} min(freq(\mathrm{i}, b1), freq(\mathrm{i}, b2)) \times w_{\mathrm{i}}}{lat(b1) + lat(b2)}$$

Here Q is set of all possible instruction types available in the instruction set (*i.e.* LLVM-IR opcodes). $lat(b)$ is the static latency of basic block which can be calculated by summing the latencies of all instructions in b. w_{i} is the latency of instruction type i. The idea here is to approximate the percentage of instruction cycles that can be saved by melding the instructions in $b1$ and $b2$ assuming a best-case scenario (*i.e.* all common instructions in $b1$

and $b2$ are melded regardless of their order). For example, two basic blocks with identical opcode frequency profile will have a profitability value 0.5.

Because meldable subgraphs are isomorphic, there is a one-to-one mapping between basic blocks (*i.e.* corresponding basic blocks). For example, in Figure 2.2 case ① the basic block mapping for CFGs L and M are $\{(\%C, \%P), (\%E, \%Q), (\%D, \%R)\}$. Assume the mapping of basic blocks in $S1$ and $S2$ is denoted by O. Subgraph melding profitability MP_S of subgraphs $S1$ and $S2$ is defined in terms of melding profitabilities of their corresponding basic blocks.

$$MP_S(S1, S2) = \frac{\sum_{(b1,b2)\in O} MP_B(b1, b2) \times (lat(b1) + lat(b2))}{\sum_{(b1,b2)\in O} lat(b1) + lat(b2)}$$

Similar to MP_B, MP_S measures the percentage of instruction cycles saved by melding two SESE subgraphs. This metric is an over-approximation, however it provides a fast way of measure the melding profitability of two subgraphs that works well in practice. We use MP_S as the scoring function for subgraph alignment.

Instruction Alignment: Notice that our subgraph melding profitability metric (*i.e.* MP_S) prioritizes subgraph pairs that have many similar instructions in their corresponding basic blocks. Therefore when melding two corresponding basic blocks we must ensure that maximum number of similar instructions are melded together. This requires computing an alignment of two instruction sequences such that if they are melded using this alignment, the number of instruction cycles saved will be maximal. We use the approach used in Branch Fusion [11] to compute an optimal alignment for two instructions sequences. In this approach compatible instructions are aligned together and instructions with higher latency are prioritized to be aligned over lower latency instructions. Compatibility of two instructions for melding depends on a number of conditions like having the same opcode and types of the operands being compatible. We used the criteria described by Rocha et al. [56] to determine this compatibility. This instruction alignment model uses a gap penalty for unaligned instructions because extra branches needs to be generated to conditionally execute these unaligned instructions. Our melding algorithm does not depend on the sequence alignment algorithm used for instruction alignment computation. We use Smith-Waterman

algorithm [54] to compute the instruction alignment because prior work [11] has shown its effectiveness. Figure 2.3a shows the instruction alignment computed for two basic blocks A and B. Aligned instructions are shown in green and instructions aligned with a gap are in red.

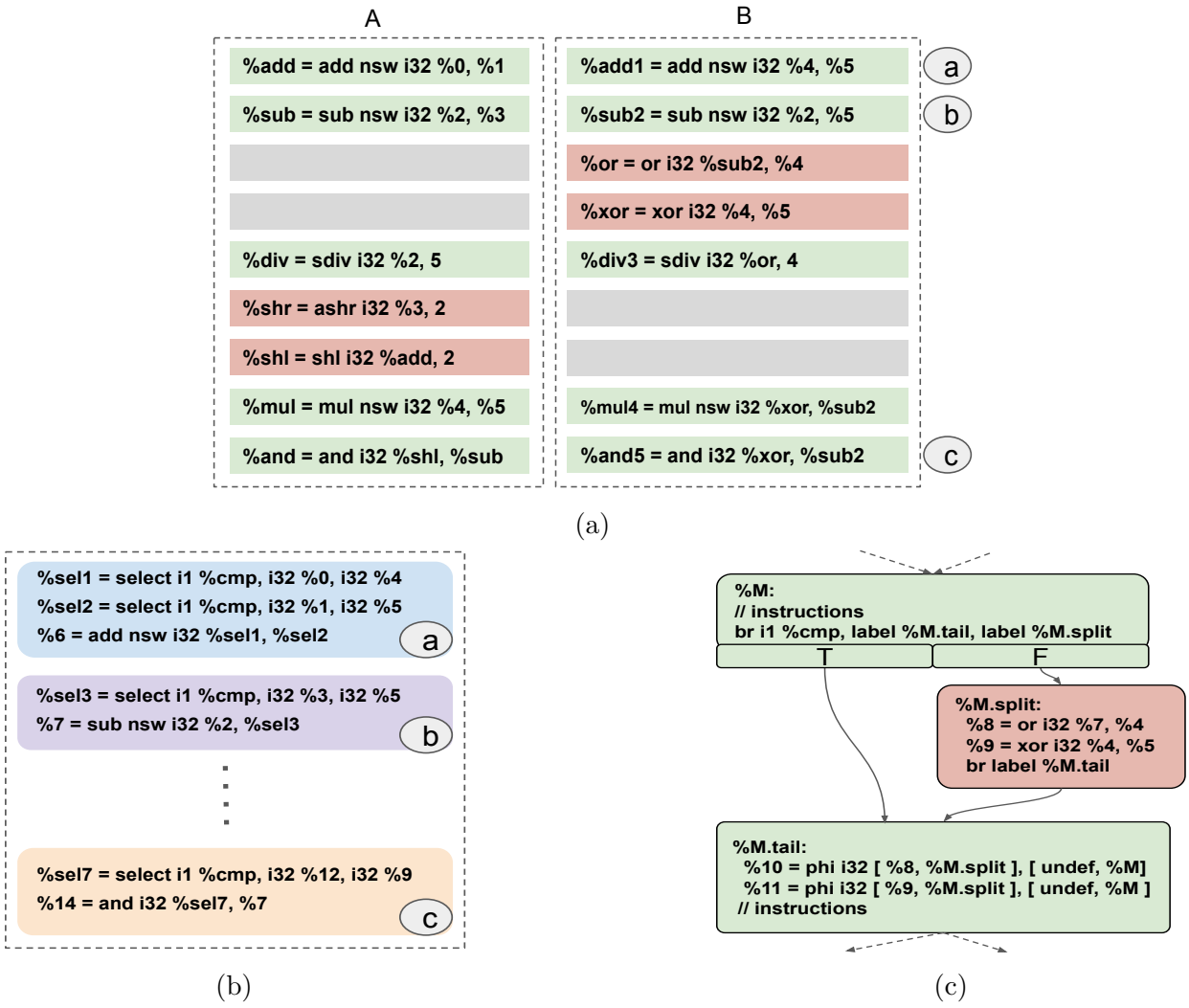

Figure 2.3. (a) Instruction alignment result for two basic blocks A and B, (b) Code generated by DARM for aligned instructions ⓐ, ⓑ and ⓒ in Figure 2.3a, (c) Unpredication applied to the unaligned instructions of basic block B in figure 2.3a

Algorithm 1: DARM Algorithm

```
Input: SPMD function F
Output: Melded SPMD function F_out
do
    changed ← false
    for BB in F do
        R, C ← GetRegionFor(BB)
        if IsMeldableDivergent(R) then
            SimplifyRegion(R)
            A ← ComputeSubgraphAlignment(R)
            for (S_T, S_F, profit) in A do
                if profit ≥ threshold then
                    Meld(S_T, S_F, C)
                    changed ← true
                end
            end
        end
        if changed then
            SimplifyFunction(F)
            RecomputeControlFlowAnalyses(F)
            break
        end
    end
while changed;
```

2.4.4 DARM Code Generation

DARM's control-flow melding procedure is shown in algorithm 1. This algorithm takes in a SPMD function F and iterates over all basic blocks in F to check if the basic block is an entry to a meldable divergent region (R) according to the conditions in Definition 2.4.5. We use *Simplify* to convert all subregions inside R in to simple regions.

We compute the optimal subgraph alignment for the two sequences of subgraphs in the true and false paths of R. We meld each subgraph pair in the alignment if the melding profitability is greater than some threshold. Subgraph melding changes the control-flow of F. Therefore we first simplify the control-flow (using LLVM's *simplifycfg*) and then recompute the control-flow analyses (*e.g.* dominator, post-dominator and region tree) required for the

melding pass. We apply the melding procedure on F again until no profitable melds can be performed.

Algorithm 2: SESE Subgraph melding Algorithm

```
Input: SESE subgraphs S_T,S_F, Condition C
Output: Melded SESE subgraph S_out
List blockPairs ← Linearize(S_T, S_F)
List A ← empty
for (B_T, B_F) in blockPairs do
    List instrPairs ← ComputeInstrAlignment(B_T, B_F)
    A.append(instrPairs)
end
PreProcess(S_T, S_F)
Map operandMap ← empty
for P in A do
    I_melded ← Clone(P)
    Update(operandMap, I_melded, P)
end
for P in A do
    SetOperands(P, operandMap, C)
end
RunUnpredication()
RunPostOptimizations()
```

Algorithm 2 shows the procedure for melding two subgraphs S_T and S_F. C is the branching condition of the meldable divergent region containing S_T and S_F. First the two subgraphs are linearized in pre-order to form a list of corresponding basic block pairs. Processing the basic blocks in pre-order ensures that dominating definitions are melded before their uses. For each basic block pair in this list we compute an optimal alignment of instructions. Each pair in the alignment falls into two categories, *I-I* and *I-G*. I-I is a proper alignment with two instructions and I-G is an instruction aligned with a gap. Our alignment makes sure that in a match the two instructions are always meldable into one instruction (*e.g.* a *load* is not allowed to align with a *store*). First we traverse the alignment pair list and clone the aligned instructions. For I-I pairs, we clone a single instruction because they can be melded. During cloning, we also update the *operandMap*, which maintains a mapping between aligned and melded LLVM values. We perform a second pass over the instruction alignment to set the operands of cloned instructions (*SetOperands*). Assume we are processing an I-I pair

with instructions I_T, I_F and cloned instruction is I_{melded}. For each operand of I_{melded}, the corresponding operands from I_T and I_F are looked up in $operandMap$ because an operand might be an already melded instruction. If the resultant two operands from I_T and I_F are the same, we just use that value as the operand. If they are different, we generate a *select* instruction to pick the correct operand conditioned by C. For an I-G pair, operands are first looked up in $operandMap$ and the result is copied to I_{melded}. Consider the instruction alignment in figure 2.3a. Figure 2.3b shows the generated code for aligned instruction pairs ⓐ, ⓑ and ⓒ. In case ⓐ, two select instructions are needed because both operands maps to different values (%0, %4 and %1, %5). In case ⓑ, the first operand is the same (%2) for both instructions, therefore only one select is needed. In case ⓒ, both first and second operands are different for the two instructions. However the second operands map to same melded instruction %7, so only one select is needed. Note that $\%cmp$ is the branching condition for the divergent region, and we use that for selecting the operands.

Melding Branch Instructions of Exit Blocks: Setting operands for branch instructions in subgraph exit blocks is slightly different than that for other instructions. Let B_T^E,B_F^E be the exit blocks of S_T and S_F. Successors B_T^E,B_F^E can contain ϕ nodes. Therefore we need to ensure that successors of B_T^E and B_F^E can distinguish values produced in true path or false path. To solve this we move the branch conditions of B_T^E and B_F^E in to newly created blocks $B_T^{'}$ and $B_F^{'}$. Now we can conditionally branch to $B_T^{'}$ and $B_F^{'}$ depending on C. For example, in Figure 2.5c basic blocks $\%M$ and $\%N$ are created when when melding the exit branches of $\%X1$ and $\%X2$ in figure 2.5b. Any ϕ node in $\%G$ (figure 2.5c) can distinguish the values produced in true or false path using $\%M$ and $\%N$.

Melding ϕ Nodes : In LLVM SSA form ϕ nodes are always placed at the beginning of a basic block. Even if the instruction alignment result contains two aligned ϕ nodes we can not meld them into a single ϕ node because *select* instructions can not be inserted before them. Therefore we copy all ϕ nodes into the melded basic block and set the operands for them using the $operandMap$. This can introduce redundant ϕ nodes which we remove during post-processing.

2.4.5 Unpredication

In our code generation process, unaligned instructions are inserted to the same melded basic block regardless of whether they are from true or false paths (*i.e.* fully predicated). This can introduce overhead due to several reasons. If the branching conditions C is biased towards the true or false path, it can result in redundant instruction execution. Also full predication of unaligned store instructions require adding extra loads to makes sure correct value is written back to the memory. *Unpredication* splits the melded basic blocks at gap boundaries and moves the unaligned instructions into new blocks. Figure 2.3c shows unpredication applied to the unaligned instructions of basic block B in Figure 2.3a. The original basic block is split to two parts ($\%M$ and $\%M.tail$) and unaligned instructions (%8 and %9) are moved to a new basic block, $\%M.split$. ϕ nodes ((%10 and %11)) are added to $\%M.tail$ to ensure unaligned instructions dominate their uses. %8 and %9 are never executed in the true path, therefore ϕ nodes' incoming values from block $\%M$ are undefined (*LLVM undef*). Note that in region replication (Section 2.4.3) we apply unpredication only to the melded basic blocks. Store instructions outside the melded blocks are fully predicated by inserting extra loads.

2.4.6 Pre- and Post-Processing Steps

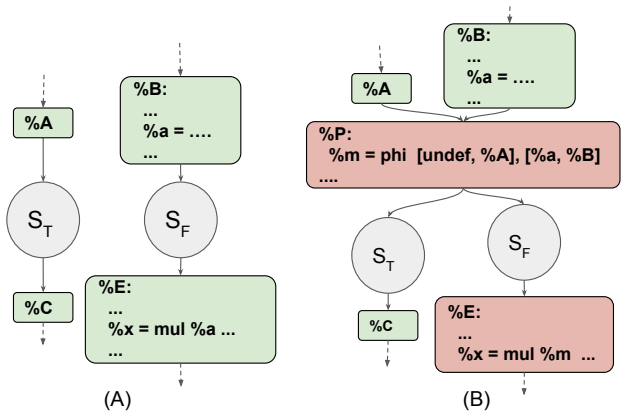

Figure 2.4. DARM pre-processing example

In SSA form, any definition must dominate all its users. However DARM's subgraph melding can break this property. Consider the two meldable subgraphs S_T, S_F in figure 2.4 Ⓐ. Definition $\%a$ dominates its use $\%x$ before the melding. However if S_T and S_F are melded naively then $\%a$ will no longer dominate $\%x$. To fix this we add a new basic block $\%P$ with a ϕ node $\%m$. All uses of $\%a$ are replaced with $\%m$ (Figure 2.4 Ⓑ). Notice that value $\%m$ is never meant to be used in the true path execution. Therefore it is undefined in true path (*undef*). We apply this preprocessing step before the melding (*PreProcess* in Algorithm 2).

Subgraph melding can introduce branches with identical successors, ϕ nodes with identical operands and redundant ϕ nodes. *RunPostOptimizations* in Algorithm 2 removes these redundancies.

2.4.7 Putting All Together

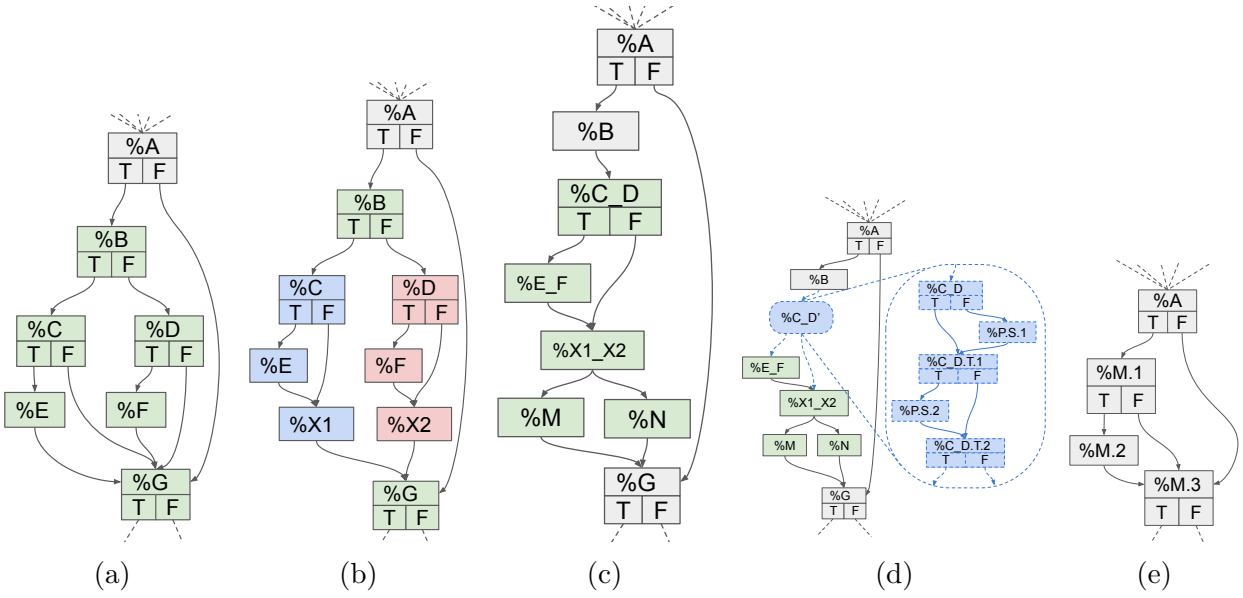

Figure 2.5. DARM melding algorithm applied to bitonic sort (Figure 2.1) (a) Original control-flow graph, (b) Region simplification, (c) DARM subgraph melding, (d) Unpredication, (e) Final optimized control-flow graph

Figure 2.5 shows how each stage of the pipeline of subgraph-melding transforms the CFG of bitonicSort kernel. The original CFG is shown in Figure 2.5a. Region ($\%B$, $\%G$) is a

meldable divergent region. Figure 2.5b shows the CFG after region simplification. Subgraphs $(\%C, \%X1)$ and $(\%D, \%X2)$ are profitable to meld according to our analysis. Figure 2.5c shows the CFG after subgraph-melding. The result after applying unpredication is shown in Figure 2.5d. Notice that the unpredication splits the basic block $\%C\_D$ (in Figure 2.5c) into 5 basic blocks (zoomed in blue-dashed blocks in Figure 2.5d). Basic blocks $\%P.S.1$ and $\%P.S.2$ are the unaligned groups of instructions and they are executed conditionally. Figure 2.5e shows the final optimized CFG after applying post optimizations. Note that *ROCm HIPCC* compiler applied *if-conversion* aggressively. Therefore the effect of unpredication step is nullified in this case.

Figure 2.5 only shows how DARM transformation changes the CFG of our running example. It does not show the change of instructions inside these basic blocks. We use Figure 2.6 to explain the generation of melded instructions for the running example. Figure 2.6a shows the LLVM-IR of the meldable divergent region ($(\%B, \%G)$ in Figure 2.5b) in our running example. During DARM code generation, basic blocks in subgraphs $(\%C, \%X1)$ and $(\%D, \%X2)$ are linearized to compute the instruction alignment. Computed instruction alignment is shown in Figure 2.6b. Notice that $[\%C, \%D], [\%E, \%F], [\%X1, \%X2]$ are the corresponding basic block pairs. In this example all instructions perfectly align with each other except for the compare instructions in basic blocks $\%C$ and $\%D$ (shown in red in Figure 2.6b). Figure 2.6c shows the LLVM-IR after applying subgraph melding and unpredication (similar to Figure 2.5d). Note that instructions %34 and %31 (compare instructions) are unaligned. Therefore unpredication step introduced basic blocks $\%P.S.1$ and $\%P.S.2$ to execute them conditionally based on the divergent condition %16. Extra ϕ instructions $\%phi.1$ and $\%phi.2$ are inserted to ensure def-use chains are not broken during the unpredication step. Out of the all aligned instructions only the branch instructions at the end of basic blocks $\%C$ and $\%D$ require select instructions during instruction-melding. For example the store instructions in basic blocks $\%E$, $\%F$ use matching operands, therefore can be melded without adding selects. On the other hand, conditional branch instructions uses values %34 and %31 and select instruction %37 is inserted (Figure 2.6c) to pick the branching condition *conditionally*. Note that the values %34 and $31 will flow to their users via the ϕ nodes $\%phi.1$ and $\%phi.2$ respectively. Therefore the select instruction (*i.e.* %37) uses these ϕ nodes as its operands.

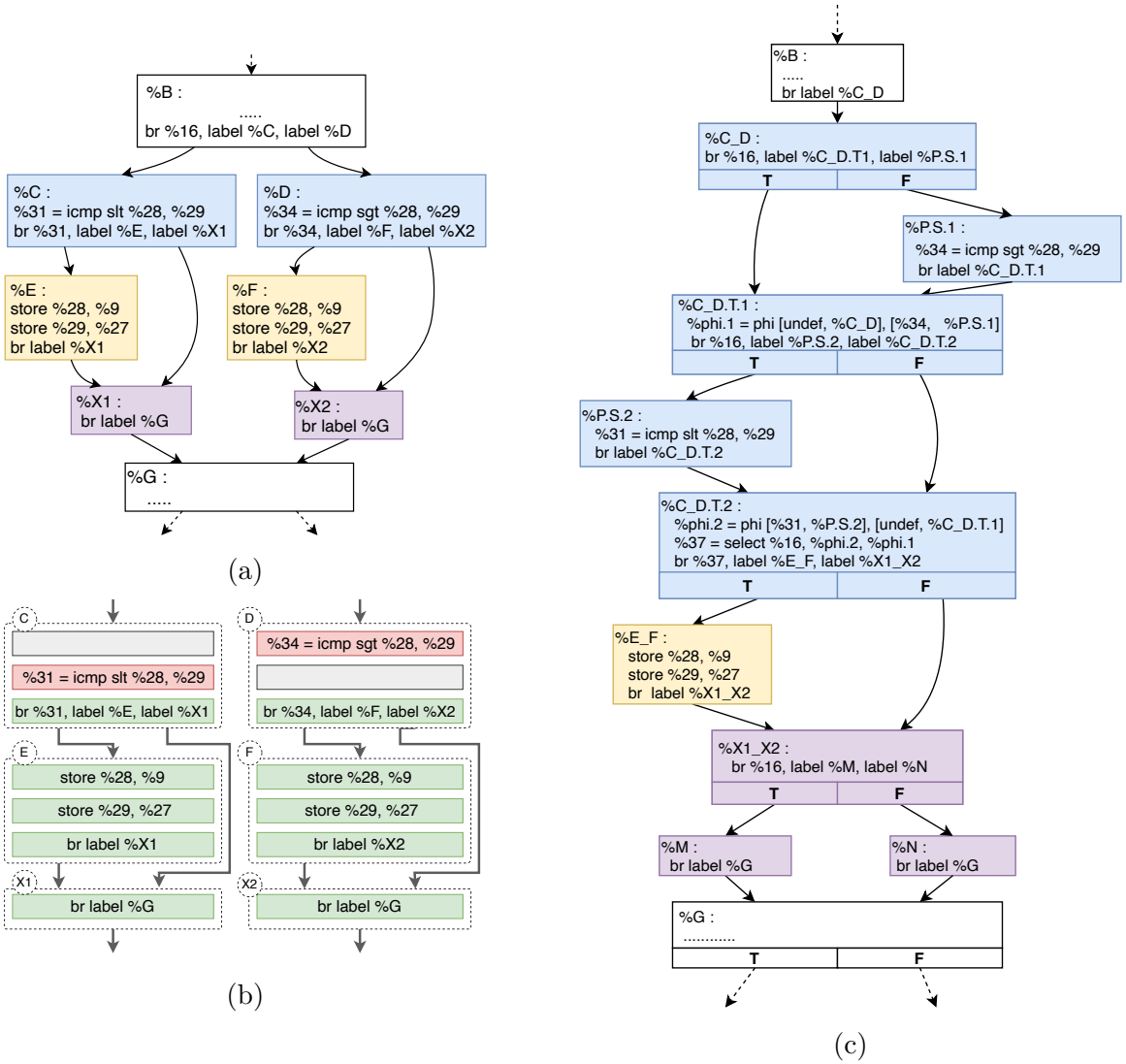

Figure 2.6. LLVM-IR before and after applying DARM transformation to our running example (a) meldable divergent region (b) instruction alignment (b) LLVM-IR generated after subgraph melding and unpredication

2.5 Implementation

We implemented the DARM algorithm described in Section 2.4 as an LLVM-IR analysis and transformation pass on top of the *ROCM HIPCC*[3] GPU compiler [40]. Both the analysis and transformation are function passes that operate on GPGPU functions. The analysis pass

[3]↑LLVM version 12.0.0, ROCm version 4.2.0

first detects meldable divergent regions using LLVM's divergence analysis. Then it finds all the profitable subgraph pairs that can be melded. We use a default melding profitability threshold of 0.2 (algorithm 1). We also provide a sensitivity analysis on this threshold in Section 2.6.5. For the instruction alignment computation, Smith-Waterman [54] algorithm was used. As described in Chapter 2.4 instructions are prioritized during the alignment based on their latency. Determining the static latency of an instruction at IR-level is a difficult problem [57]. We use modified version of LLVM cost model [58] to obtain instruction latencies for melding profitability and instruction alignment computations. The transformation uses the output of analysis to perform DARM's code generation procedure (Section 2.4.4). The transformation pass also performs the unpredication, pre- and post-processing steps described in Sections 2.4.5 and 2.4.6. LLVM pass is implemented in $\sim$ 2500 lines of C++ code. In order to produce the program binary with our pass, we had to include our pass in the *ROCM HIPCC* compilation pipeline.

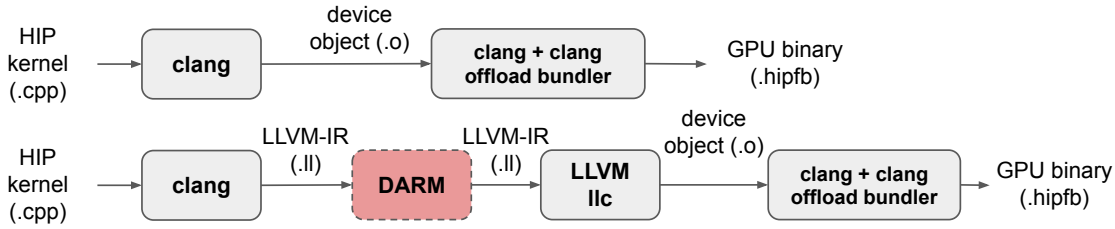

Figure 2.7. Modifications made to compilation flow

Most GPGPU compilers (*e.g.* CUDA `nvcc`, ROCm `HIPCC`) use *separate compilation* for GPU device and CPU host codes. In separate compilation, the GPU device code and CPU host code are compiled separately. Final executable contains the device binary embedded in the host binary. Figure 2.7 shows the default host code compilation pipeline and modifications (shown in red) we did on the pipeline to add our pass. In the default pipeline, device code is first compiled into an object file using `clang`. This object file is then converted to a special GPU binary format (`hipfb`) using `clang-offload-bundler` and embedded with the host binary. In modified compilation pipeline, we did the following changes. First we added a switch in LLVM-IR optimizer (*i.e.* `opt`) to enable our pass. We modified the original pipeline by adding commands to compile host code to LLVM-IR. We run our pass on top of

the host LLVM-IR and transform it. Finally, we use LLVM's static compiler (`llc`) [59] to compile the IR into a device object file. Our pass runs only on device functions and avoids any modifications to host code. The rest of the compilation flow is as same as the one without any modification. These include the host code compilation commands and commands used for linking. We developed a `pyhton` script to automate the generation of these modified compilation commands. This script takes in the original set of compilation commands used to compile the `HIP` program (obtained using `-###` flag in `clang`) and emits a modified set of commands with changes described above.

2.6 Evaluation of DARM

2.6.1 Evaluation Setup and Benchmarks

We evaluate the performance of DARM on a machine with a *AMD Radeon Pro Vega 20* GPU. This GPU has 16 *GB*s of global memory, 64 *kB* of shared memory (*i.e.* Local Data Share (LDS)) and 1700 *MHz* of max clock frequency. The machine consists of *AMD Ryzen Threadripper 3990X* 64-Core Processor with 2900 *MHz* max clock frequency.

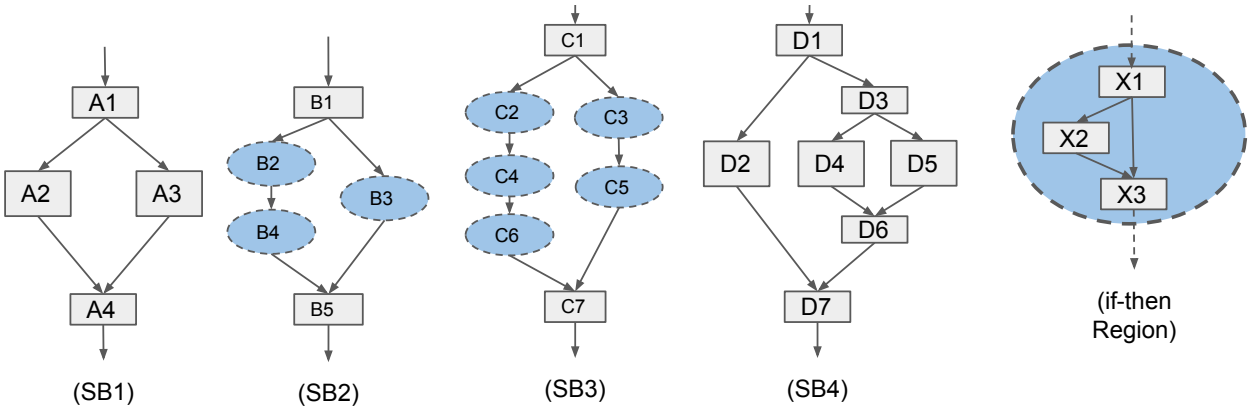

Figure 2.8. Control-flow patterns in synthetic benchmarks. Square: basic block and Circle: *if-then* region (shown on right)

We use two different sets of benchmarks. First, to assess the generality of DARM, we create several synthetic programs that exhibit control divergence of varying complexity. While many real-world programs are hand-optimized to eliminate divergence, these synthetic programs both qualitatively demonstrate the generality of DARM over prior automated

divergence-control techniques, and show that DARM can automate the control flow melding that would otherwise have to be done by hand.

Synthetic Benchmarks

Each synthetic kernel consists of two nested loops. The inner loop contains a divergent region with different control-flow structures (SB1, SB2, SB3 and SB4 in Figure 2.8). Every divergent path computes on different pieces of data from shared memory. SB1 has simple diamond-shaped control-flow with basic blocks A2 and A3 performing identical computations. In SB2 and SB3; circled regions are *if-then* sections. *Then* blocks in region pairs B2-B3 (in SB2), C2-C3 and C6-C5 (in SB3) consist of identical computations. In three-way divergent kernel SB4, basic blocks D2, D4, and D5 are performing identical computations. Basic blocks/regions with identical computations have high melding profitability. Synthetic benchmarks SB1-R, SB2-R, SB3-R and SB4-R have same control-flow structure as SB1-SB4 but contain non-identical computations in the basic blocks. All synthetic benchmarks copy the input variables into shared memory, perform the computation, and write back again to global memory. We used randomly generated arrays of size 2^{20} for each input variable.

Prior control-flow melding techniques (tail merging [13] and branch fusion [11]) cannot meld the full set of synthetic benchmarks. Tail merging can combine the divergent *if-then-else* blocks in SB1 and SB4 but cannot fully merge divergent regions. It cannot merge the -R variants due to the different instructions in the divergent paths. Branch fusion subsumes tail merging, and can fully merge *if-then-else* blocks in SB1, SB4 and their -R variants. However, it cannot be applied to the more complex control flow of SB2 and SB3, or their -R variants. In SB4, iterative application of branch fusion can meld blocks D4,D5 and D2. However, its -R variant can not be fully melded by branch fusion due to non-identical computations being un-predicated (*cf* Section 2.4.5). In contrast, DARM melds it by using *region replication* (*cf* Section 2.4.3).

Real-world Benchmarks

Second, to show DARM's effectiveness on real-world programs, we consider 7 benchmarks written in *HIP* [35]. These benchmarks were taken from well-known highly hand-optimized GPU benchmark suites or optimized reference implementations of papers. We selected these benchmarks because they contain divergent if-then-else regions that present melding op-

portunities for DARM. We do not consider benchmarks that do not present any melding opportunities for DARM because they are not modified by DARM in any way.

Bitonic Sort (BIT) Our running example is bitonic sort [50]. In this kernel, each thread block takes in a bucket and performs parallel sort. We used an input of 2^{26} elements and varied the bucket (*i.e.* block) size. Branch fusion cannot handle the control-flow in BIT.

Partition and Concurrent Merge (PCM) PCM is a parallel sorting algorithm based on Batcher's odd-even merge sort [60]. PCM performs odd-even merging of *buckets* of sorted elements at every position of the array leading to loops with nested data-dependent branches. We used an array of 2^{28} elements with different number of buckets. PCM's control-flow is too complex for Branch fusion to merge.

Mergesort (MS) A parallel bottom-up merge sort implementation. The kernel has data-dependent control-flow divergence in the merging step. We used an input array with 2^{20} elements. Merge sort has simple diamond control flow, so can be handled by branch fusion.

LU-Decomposition (LUD) LUD implementation from the Rodinia benchmark suite [61]. We focus our evaluation on the *lud_perimeter* kernel in this benchmark. *lud_perimeter* contains multiple divergent branches that depend on thread ID and block size. We use a randomly generated matrix of size 16384×16384 as the input. Branch fusion can successfully merge divergent control-flow in LUD when the loop is unrolled.

N-Queens (NQU) N-Queens solver uses backtracking to find all different ways of placing N queens on a NxN chessboard without attacking each other. We have used the kernel from the GPGPU-sim benchmark suite [62] with N is 15.

Speckle Reducing Anisotropic Diffusion (SRAD) SRAD is diffusion based noise removal method for imaging applications from Rodinia benchmark suite [61]. We have used an image of size 4096×4906 as input.

DCT Quantization (DCT) An in-place quantization of a discrete cosine transformation (DCT) plane [63]. The quantization process is different for positive and negative values resulting in data-dependent divergence. We use a randomly generated DCT plane of size $2^{15} \times 2^{15}$ as input. Branch fusion can handle the control flow of DCT.

Baseline and Branch Fusion: Our baseline implementations of these kernels have been hand-optimized (except, obviously, for optimizations that manually remove control divergence by applying DARM-like transformations). This optimization includes using shared memory when needed to improve performance. The baseline implementations were compiled with `-O3`. Branch fusion [11] was implemented in the Ocelot [64] open-source CUDA compiler that is no longer maintained and does not support AMD GPUs. We implemented branch fusion by modifying DARM to apply melding for diamond-shaped control-flow (*if-then-else*). We use this for comparison against branch fusion. Branch fusion cannot fully handle the control-flow of BIT, PCM, and NQU. Loop unrolling enables successful branch fusion in LUD.

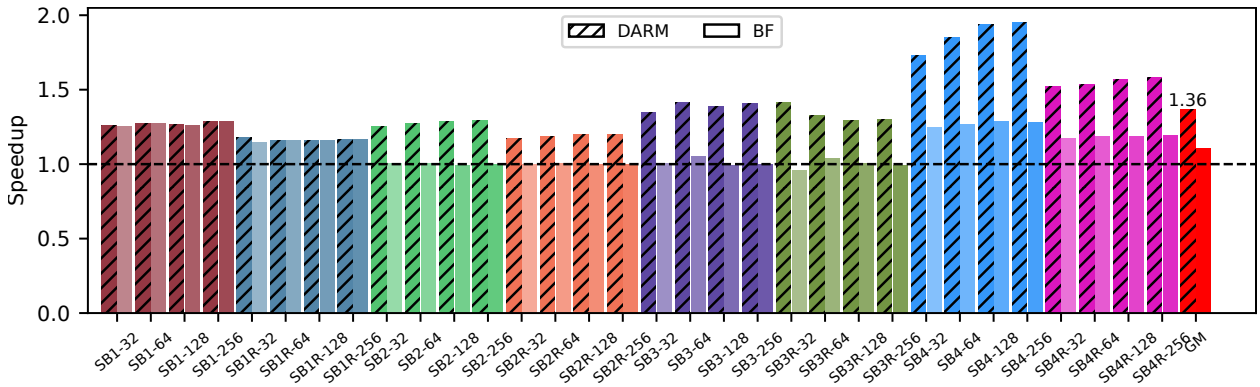

Figure 2.9. Micro Benchmark Performance. GM is geomean of DARM's speedup over baseline.

Block Size: Each of these kernels has a tunable *block size*—essentially, a tile size that controls the granularity of work in the inner loops. Because the correct block size can be dependent on many parameters (though for a given input and GPU configuration, one is likely the best), our evaluation treats block size as exogenous to the evaluation, and hence considers behavior at different block sizes for each kernel. In other words, our evaluation asks: if a programmer has a kernel with a given block size, what will happen if DARM is applied?

Note that of these kernels, only LUD exhibit divergence that depends on block size. This means that all the other benchmarks will experience divergence regardless of block size.

LUD's divergence, on the other hand, is block size dependent. For some block sizes, the kernel will be divergent, while for others, it will be convergent.

2.6.2 Performance

Figure 2.9 shows the speedups for the synthetic benchmarks with different block sizes. DARM can successfully meld all 4 control-flow patterns we consider in the synthetic benchmarks and gives a superior performance than the baseline and branch fusion (geo-mean speedups of 1.36× for DARM and 1.10× for branch fusion over the baseline). The performance for random (-R) variants are slightly lower for each of the patterns. This is because -R variants contain random instruction sequences and instructions do not align perfectly, causing DARM to insert *select* instructions and branches to unpredicate unaligned instruction groups. Speedups observed for SB3 and SB3-R are better than SB1, SB2 and their -R variants because DARM melds multiple subgraph pairs in the SB3 control-flow pattern (Figure 2.8) and control-flow divergence is reduced more in this case. We observe the highest performance improvement for SB4 and SB4-R because DARM melds basic blocks D2, D4, and D5 (Figure 2.8) using *region replication*. SB4 and its -R variant have 3-way divergence because of the *if-else-if-else* branch. Applying *region replication* along with subsequent simplification passes greatly reduces this original three-way divergence.

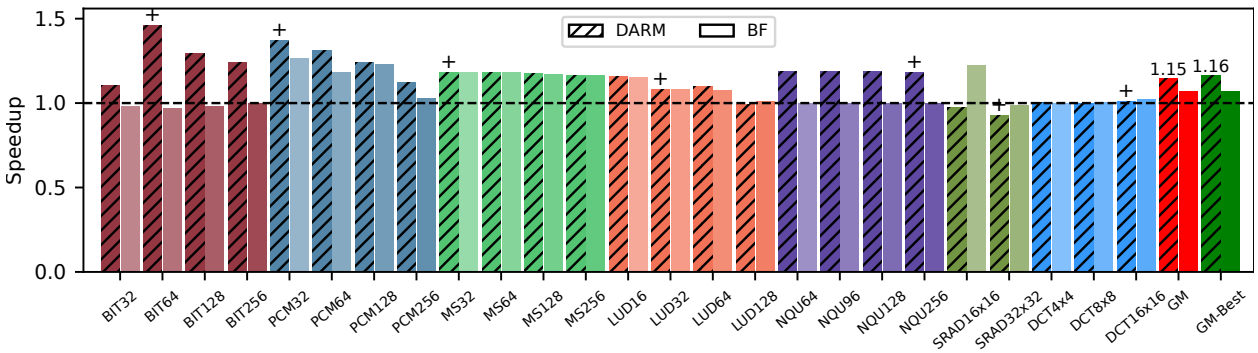

Figure 2.10. Real-world Benchmark Performance. + marks block size with best baseline runtime. GM is geo-mean of DARM's speedup on all benchmarks; GM-Best is DARM's speedup on + configurations.

Figure 2.10 shows the speedups for real benchmarks DARM always improves the performance (1.15× geo-mean speedup over all benchmarks and 1.16× geo-mean speedup over the best baseline variants) except for SRAD (see below). The highest relative improvement in performance can be seen in BIT and PCM for all block sizes. This is because both these benchmarks are divergent regardless of the block size and they have complex control-flow regions with shared memory instructions. DARM successfully melds these regions and reduces divergence significantly. Branch fusion improves performance in PCM by melding *if-then-else* blocks. In LUD, the divergence is block size dependent, and the kernel is divergent only at block sizes 16, 32 and 64, where we see a visible performance improvement introduced by DARM. NQU contains a time-consuming loop with divergent *if-then-elseif-then* section. DARM applies *region replication* to remove divergence, achieving superior performance. SRAD kernel has both block size-dependent and data-dependent divergent regions (say R_B and R_D respectively). Both R_B and R_D consists of *if-then-else–if-then-else* chains. R_B contains no shared memory instructions and melding does not improve performance (for both DARM and branch fusion). However R_D contains a 3-way divergent branch with shared memory instructions and the divergence is biased *i.e.* execution only takes 2 of the 3 ways. In this case branch fusion has better performance at block size 16, because blocks that get melded happen to be on the divergent paths. However DARM has more melding options than branch fusion, and it melds all 3 paths adding extra overhead. At block size 32, the extra overhead introduced by melding R_B becomes significant and both DARM and branch fusion exhibit a performance drop. Performance drop for DARM can be avoided by prioritizing the melding order (*i.e.* apply melding to divergent regions with most profitable subgraphs first). However, prioritizing melding order is not considered in this work.

In most cases (except SRAD), the block size for best performing baseline is also the one that gives the best absolute performance for DARM. Interestingly, for 4/7 benchmarks (BIT, PCM, MS, and DCT), not only does this best baseline block size produce the best absolute DARM performance, it also produces the best *speedup* relative to the baseline: the block size that makes the baseline perform the best, actually exposes more optimization opportunities to DARM.

We use *rocprof* [65] to collect ALU utilization and memory instruction counters to reason about performance. We focus on the block sizes for each benchmark where DARM has highest improvement over the baseline.

2.6.3 ALU Utilization

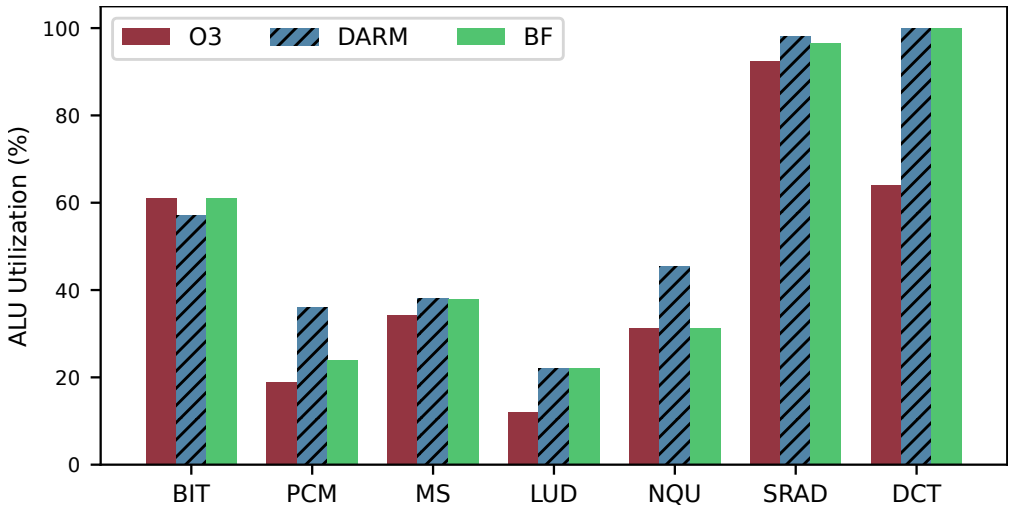

Figure 2.11. ALU Utilization.

DARM's melding transformation enables the ALU instructions in divergent paths to be issued in the same cycle. This effectively improves the SIMD resource utilization. Figure 2.11 shows the ALU utilization (%). As expected DARM improves the ALU utilization significantly for most benchmarks. In BIT, divergent paths does not have common comparison operators ($>$ and $<$ comparisons in lines 9 and 13 in Figure 2.1). Even though DARM unpredicates these instructions, later optimization passes decide to fully-predicate them resulting in lower ALU utilization.

2.6.4 Melding of Memory Instructions

Figure 2.12 shows the normalized number of global and shared memory (*i.e.* local data share) instructions issued after applying DARM. In LUD, there are many common shared memory instructions in divergent paths. However these instructions do not have different memory alignments, therefore cannot be melded into a single instruction. Unpredicated

shared memory instructions are predicted by other optimization passes in LLVM resulting in higher instruction count. Melding reduces the global memory instruction count in LUD. DCT does not have any memory instructions in the divergent region and does not use shared memory. In BIT and PCM, the melded regions contain a lot of shared memory instructions. Therefore the reduction in shared memory instructions is significant and correlate with the performance gain. We find that melding shared memory instructions is more beneficial than melding ALU instructions because shared memory instructions have higher latency than most ALU instructions, though lower latency than global memory instructions. Therefore there is 2× improvement in cycles spent if two divergent shared memory instructions are issued in the same cycle.

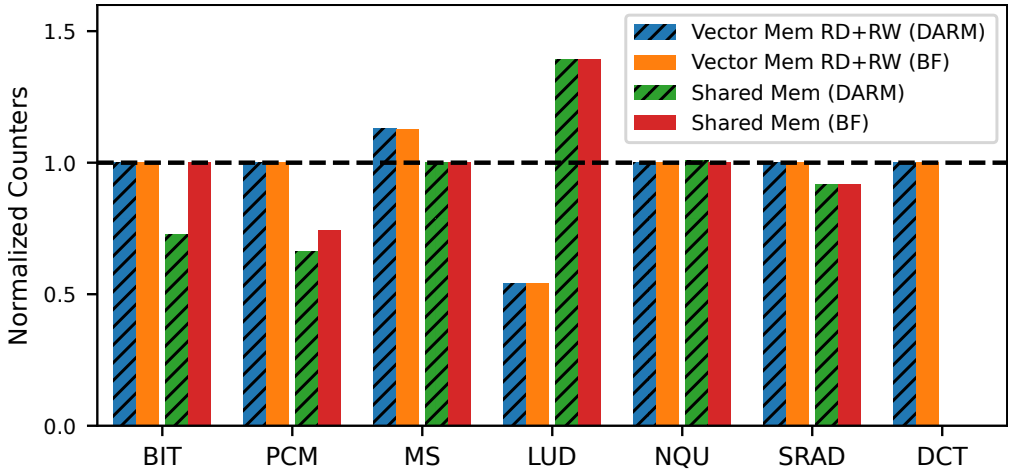

Figure 2.12. Normalized Memory Instruction Counters.

Counterintuitively, melding may be less effective for global memory instructions, despite their longer latency. If both accessed values are not in the GPU cache, then memory controller can often fulfill the requests in the same cycle regardless of whether they were issued in same cycle or not. In that case DARM does not change the performance of the program. This is visible in LUD, where the number of shared memory instructions has gone up and global memory instructions have gone down, but we do not observe a sizable improvement in performance. The reason for this coalescing of memory accesses is fairly subtle, and deserves unpacking. Modern GPUs feature *Independent Thread Scheduling* (ITS) [7], [66], where divergent threads from the same warp can be scheduled independently when execution stalls.

Hence, when one group of threads suffers a miss, the GPU can schedule other threads *from the same warp*. These threads will then reach their corresponding access, which will be coalesced by the memory controller. So the total latency experienced by both groups of threads is one miss, identical to if DARM melded the accesses statically. Older GPUs that did not feature ITS would likely still see a benefit from DARM.

2.6.5 Melding Profitability Threshold

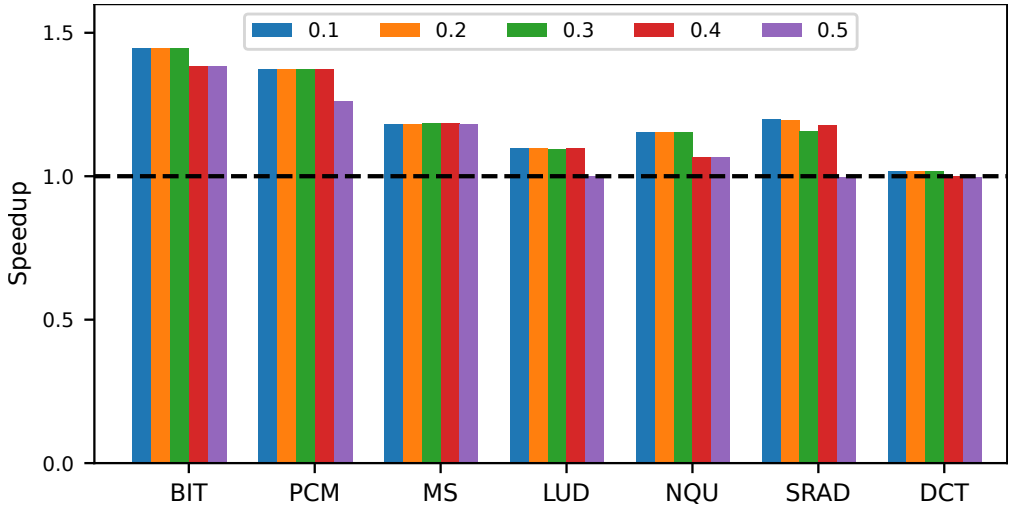

Figure 2.13. Variation of melding profitability thresholds.

Figure 2.13 shows the performance of DARM for different melding profitability thresholds on the real-world benchmarks considering DARM's best performing block sizes. For all benchmarks, we observe that DARM's speedup reduces as we increase the threshold due to lost opportunities.When we reduce the threshold, increment in the improvement of the performance of DARM becomes insignificant (after 0.2). But we cannot reduce it to zero because every possible pair would be melded and the subsequent CFG simplification passes would unpredicate them. This may drive the pass pipeline into an infinite loop and makes DARM non-convergent.

2.6.6 Compile Time

Table 2.2 shows the device code compilation times for the baseline and DARM. We omit the time for compiling host code and linking because it is constant for both the baseline and DARM. Since we perform the analysis and the instruction alignment – the most costly parts – at the basic block level rather than performing at a higher level (*i.e.* function or region level), we incur negligible compilation overhead. Compilation time overhead introduced by DARM is a small fraction of total compilation time (including host code) for all cases.

Table 2.2. Average Compile Time (s)

Benchmark	O3	DARM	Normalized
BIT	0.4804	0.5018	1.0444
PCM	0.5690	0.5942	1.0443
MS	0.8037	0.8064	1.0035
LUD	0.5993	0.6294	1.0502
NQU	0.4687	0.4738	1.0109
SRAD	0.4999	0.5121	1.0244
DCT	0.4398	0.4439	1.0093

DARM's compile time depends on the size of basic blocks that get melded and the structure of the program since it determines different types of melding opportunities. A slight overhead in compilation time of LUD is caused by sequence alignment overhead on large basic blocks (created by loop unrolling). PCM and BIT have divergent regions inside an unrolled loop, therefore DARM's meldable subgraph detection incurs overhead.

2.6.7 Types of Melding

Table 2.3 provides breakdown of types of melding performed by DARM on all the benchmarks for the best performing block size. Only BIT and PCM has opportunities for Region-Region melding, and only PCM, NQU, and SRAD have opportunities for Basic block-Region melding. Presence of Basic block-Region melding opportunity results in *region replication*.

Table 2.3. Different Types of Melding

Benchmark	Block Size	BB-BB	BB-Region	Region-Region
BIT	64	0	0	21
PCM	32	16	1	1
MS	32	1	0	0
LUD	16	3	0	0
NQU	64	0	2	0
SRAD	16x16	3	6	0
DCT	16x16	1	0	0

2.7 Discussion

2.7.1 General Applicability of DARM

Most of the GPGPU benchmarks are heavily hand optimized by expert developers and this often include DARM like transformations to remove control-flow divergence [11]. We evaluate DARM on limited set of real-world benchmarks mainly because of this reason. However we also emphasize that doing DARM-like transformations by hand is time-consuming and error-prone. For example, it took us several hours to manually apply control-flow melding to LUD kernel. Therefore, offloading this to the compiler can save a lot of developer effort.

The benefits of DARM is not limited to reducing control-flow divergence in GPGPU programs. DARM can be used to reduce control-flow divergence in any hardware backends and programming models that support *Single-Program-Multiple-Data (SPMD)* paradigm (*e.g.* Intel/AMD processors with ISPC [37]). DARM can be used to reduce branches in a program. This property can be exploited to accelerate software testing techniques such as symbolic execution [30]. DARM factor out common code segments within *if-the-else* regions of a program. Therefore, it can be used as an intra-function code size reduction optimization as well. Aforementioned applications of DARM suggest that it is useful as a general compiler optimization technique. We explore some of these applications in our future work.

2.7.2 Melding Non-isomorphic CFG Regions

As shown in Section 2.4.3, DARM does not meld non-isomorphic SESE regions when both regions contain more than one basic block. This precludes melding opportunities for basic blocks in non-isomorphic SESE regions. Solving this problem requires generalizing *Region Replication* to expand any given smaller subgraph to match a larger subgraph. Even though this may not be possible for all cases, performing this restructuring may unleash new opportunities to enhance performance and we leave this to future work.

2.7.3 Precision of Divergence Analysis

Applying DARM to meld a truly *non-divergent* branch may add new overhead to the program. Hence using a precise divergence analysis is important for the enhancement of performance. If a sophisticated divergence analysis is available, based on its results (*e.g.* probability of divergence), we can decide whether running our pass will or will not be beneficial.

2.7.4 Shared Memory, Cost Model and Sequence Alignment

In Section 2.6, we have shown that when shared memory is used to improve the baseline, it does not steal the opportunity from DARM to meld, because melding shared memory instructions also results in better performance than the improved baseline. Exploiting this opportunity requires maximizing the alignment of shared memory instructions which can be achieved by using a refined instruction cost model.

2.8 Related Work

2.8.1 Techniques for Reducing Control-Flow Divergence

As discussed in the previous sections, control-flow divergence can be a serious bottleneck in GPGPUs or in any hardware platform that facilitates a SPMD-style programming model. There have been many attempts to reduce the performance degradation caused by control-flow divergence both using compiler transformations and architectural enhancements in the underlying hardware.

Software Techniques: Tail Merging [13] is classic compiler optimization technique used primarily for code compaction. Tail merging works by moving the common suffix/tail (*i.e.* instructions at the end of the block) of a set of basic blocks into their common successor. Common instructions do not have to be identical in terms of operands because *select* instructions can be inserted to make them look identical. Tail-merging reduces the number of instructions in the program, hence makes the code size smaller. Tail merging also helps with control-flow divergence because it reduces the number of instructions executed on divergent paths. Note that tail merging works only when the tail-merged basic blocks have identical operation sequences at their tails i.e. the tail can be moved to the common successor up until any further matching instructions are not available in all the predecessors. As described in Chapter 2.1, Coutinho *et al.* introduced branch fusion which is a generalization of tail merging applicable only for CFGs with a diamond shaped control-flow [11]. Branch fusion uses biological sequence alignment algorithms [54], [67] to find common instructions in the `if` and `then` paths of the diamond shaped control-flow. Aligned instructions are moved to new converged basic blocks so that diverged threads can reconverge early. Unaligned instructions are executed conditionally using the original divergent branching condition. Even though this technique is more general than Tail Merging, it is constrained to diamond shaped control-flow and can not be used when more complicated control-flow structures are present.

IPDOM-based reconvergence guarantees the earliest reconvergence only for structured CFGs. However certain programming languages constructs (such as `goto` statements) and compiler optimizations can result in unstructured control-flow. Untructured control-flow is quite common in GPGPU applications [68]. When unstructured control-flow is present certain basic blocks can get executed multiple times for different threads resulting in reduced SIMT resource utilization. Anantpur and Govindarajan proposed a technique to structure the unstructured CFGs by inserting guard blocks and guard variables [9]. The execution of a given basic block is guarded by a guard variable. The value of guard variable is determined at the end of basic block based on which successor basic block must be executed next. Applying this technique converts the unstructured CFG into a series of *if-then* chains without exponentially increasing the code size. Authors shows the utility of this method by applying

it for control-flow divergence reduction, loop collapsing, branch interleaving and SIMT stack depth reduction.

Fukuhara and Takimoto proposed *Speculative Sparse Code Motion (SSCM)* to reduce divergence in GPGPU programs [69]. SSCM is an aggressive version of Sparse Code Motion [70] that hoists redundant expressions out of branches to reduce the impact of divergence. SSCM does not alter the CFG of a program. Han *et al.* proposed *Branch Distribution* which factors out identical computations inside *if-the-else* regions (*i.e.* diamond control-flow) and move them out to reduce control-flow divergence.

Collaborative Context Collection (CCC) is a software-only technique that enables improved warp execution efficiency by collecting divergent tasks in a warp and differing their execution until all the lanes can be occupied by similar tasks [12]. CCC is applicable for kernels with repetitive divergent tasks with independent iterations (*e.g.* graph algorithms like BFS). Key idea in CCC is capturing just enough information of a divergent thread's context into a *context stack*, so any other thread can continue its progress at a later time. Context is generally a subset of thread-private registers. If the collected tasks are not enough to occupy all the SIMT lanes execution moves to the next iteration. If there are enough tasks in the collection all lanes execute the divergent task. CCC enables full utilization of SIMT resources with minimal overhead. Another related technique to CCC is *Iteration Delaying* [10]. Iteration delaying can be applied when there is a divergent branch inside a loop. Instead of executing both paths of the divergent branch only a chosen path is executed in a given iteration. If a thread does not take the chosen path, its iteration is *delayed*. Two strategies are considered for the path choice, a majority vote strategy and a round-robin strategy. Round-robin strategy shows the best average case performance because it avoids thread starvation problem in majority-vote strategy. One of the drawbacks of iteration delaying is it can increase memory divergence. It should be noted that CCC is more advanced than iteration delaying because in CCC each thread's progress can be transferred to a different thread. Therefore, CCC has much more flexibility in scheduling the computation.

Recently, Damani *et al.* introduced *Speculative Reconvergence* which identifies class of divergence problems that can benefit from reconverging speculatively rather than reconverging at the IPDOM [71]. GPU programs often contains common code that can not be executed

in convergent manner because under IPDOM-based reconvergence model threads are not allowed to reconverge before the common code section. Examples include divergent *if-then* branches inside loops, inner loops with divergent trip counts, and common functions calls inside *if-then-else* branches. Speculative reconvergece allows the user to specify potential alternative reconvergence points (*e.g.* expensive divergent basic blocks or function calls) for improved SIMT efficiency. Then compiler algorithm inserts necessary synchronization operations and soft barriers to ensure the correct and convergent execution of the common code. Speculative reconvergence uses the ISA modifications introduced with *Independent Thread Scheduling* in NVIDIA Volta architecture to implement this technique. Note that in pre-Volta GPUs all the threads in a warp have common thread state information (program counter and call stack). But in Volta architecture each thread is given private thread state information enabling more fine-grained thread scheduling [66].

Common Subexpression Convergence (CSC) [72] is a similar technique to branch fusion that move common sub-expressions out of divergent code paths to reduce divergence. CSC introduces 3 code transformations *hoist, sink* and *spit* to move common sub-expressions out of divergent paths. Hoist moves the expression before the divergent branch, sink moves common sub-expressions after the earliest reconvergece point of the divergent branch. Split transformation splits divergent paths to facilitate moving common sub-expressions that can not be moved by using sink or hoist alone. CSC uses dynamic programming to find common sub-expressions in divergent paths similar to branch fusion. Complex nested control-flow is handled by using *branch flattening* that converts control dependances into data dependances (*i.e.* if-conversion).

Hardware Techniques: The control-flow divergence problem is tightly related to the design of GPGPU microarchitecture, specifically how the SPMD-style programming model is supported in hardware using the machinery described in section 2.2.1. Therefore, a lot of attempts have been made to tackle the control-flow divergence problem at a hardware level.

Contemporary GPGPUs uses IPDOM-based reconvergence model described in Chapter 2.1 to handle divergent control-flow. This is implemented using a hardware based stack called the *SIMT stack*. Each warp has its own SIMT stack that keeps track of what control-

flow path is active at a given cycle. This is also known as the *Single-Path Execution (SPE)* because only a single path is schedulable at a time. This restriction severely limits the SIMT resource utilization (*i.e.* SIMD lanes the GPU) in single-path stack model. In the literature the terms SIMT efficiency or warp execution efficiency is used interchangeably to denote the percentage of SIMT resources used by a particular execution model. Next we look at some micro-architectural enhancements proposed to avoid the limitations of SPE model.

Typically a GPGPU has thousands of schedulable warps in flight when executing a kernel. Some of these warps can diverge due to divergent control-flow. Dynamic Warp Formation (DWF) [34] forms new warps by combining threads from diverged warps such that in the newly formed warp all the SIMD lanes are occupied by the threads. DWF details the necessary register file enhancements and changes required in the warp scheduling strategy to make this technique useful. Fung *et al.* showed that DWF can exhibit pathological warp scheduling behaviors casing a situation called *starvation eddy*. In an starvation eddy, certain threads can fall behind in scheduling resulting in much lower SIMD efficiency and increased memory stalls. Authors proposed *Thread Black Compaction (TBC)* to avoid these limitations of DWF. Key idea in TBC is to exploit the control-flow locality of threads within a thread block during warp scheduling. TBC employs a block-wide reconvergence stack instead of the per-warp reconvergence stack in order to achieve this. Threads are *compacted* into new warps at diverging branches and restored to their original warp groupings when the compacted threads reach their reconvergence point. Block-wide reconvergece stack makes warp schedulers job easier and avoids the starvation eddy situation in DWF. TBC also extends the IPDOM based reconvergence stack to include *likely reconvergence points* which allows diverged threads to reconverge early.

Dynamic Warp Subdivision (DWS) [73] attempts to avoid the limitations of single-path stack by allowing execution of divergent paths to interleave. This improves the instruction level parallelism. DWS achieves this by creating independently schedulable units called *warp splits*. The progress of warp splits are tracked using a warp split table and the decision to split the warp or not at a divergent branch is taken based on heuristics. One drawback of DWS is that individual warp splits does not keep track of their earliest reconvergence point therefore fails to reconverge at the earliest opportunity. This can reduce the SIMD

resource utilization. Rhu *et al.* proposed *Dual-Path Execution (DPE)* model to alleviate the drawbacks of DWS [5]. Dual-path execution model extends SPE by allowing it to keep track of two control-flow paths concurrently. DPE adds minor overhead to the microarchitecture and retains the benefits of both SPE and DWS. ElTantawy [7] proposed *Multi-Path IPDOM* which extends the DPE model to concurrently schedule any number of divergent control-flow paths while still ensuring IPDOM-based reconvergence. Multi-path IPDOM replaces the traditional SIMT stack with two tables, *warp split table* to record all warp splits in flight and *reconvergence table* to record the reconvergence points of all warp splits. Furthermore, Multi-path IPDOM can opportunistically reconverge when unstructured control-flow is present in the CFG which not possible with the previous techniques. Rogers *et al.* showed the impact of warp size in the performance of real-world GPU applications. Certain divergent GPU applications can benefit from smaller warp size because smaller warps allows divergent control-flow paths to execute concurrently. However, convergent GPU applications can suffer performance penalties with smaller warps because smaller warps destroy the horizontal locality benefits of larger warps. To achieve best of both worlds, authors propose *Variable Warp Sizing (VWS)*. VWS uses a novel warp ganging microarchitecture that is capable of shrinking the warp size depending on the application characteristics. VWS enables divergent applications to execute multiple concurrent control-flow paths together with improved SIMT efficiency while forcing convergent applications to use wider warps to achieve the performance and energy efficiency of traditional GPU architectures.

Most of the architectural techniques described in this Section may not be feasible to be implemented on commodity GPGPUs because of the power and area overhead introduce by them. However, the microarchitecture design choices explored in these techniques have inspired many modern GPGPU designs.

2.8.2 Other Related Work

In this section, we discuss related techniques that are used in applications other than control-flow divergence reduction, but have certain commonalities with our approach. The general strategy of identifying and transforming similar code sequences is well-studied idea in

code compaction (*i.e.* code-size reduction). *Tail Merging* is a standard, but restrictive, compiler optimization used to reduce the code size by merging identical sequences of instructions. Chen *et al.* used generalized tail merging to compact matching *Single-Entry-Multiple-Exit* regions [13]. LLVM's *MergeFunction* pass efficiently identify and merge identical functions to reduce code size [74]. Recently, Rocha *et al.* has presented *Function Merging with Sequence Alignment (FMSA)*, which uses biological sequence alignment algorithms to compute similar instruction sequence between functions and, merge them to reduce code size [55]. Sequence alignment is an expensive computation. Therefore, FMSA relies on ranking heuristics to find functions that are sufficiently similar. Later this work was extended to support function merging in static single assignment (SSA) form [56]. Primary objective of these techniques is to reduce code size. Repurposing them to reduce control-flow divergence would require significant work because the effect on the structure CFG must be taken into consideration.

2.9 Conclusion

Divergent control-flow in GPGPU programs can cause significant performance degradation because thread execution needs to be serialized if threads in a warp exercise different control-flow paths. We presented DARM, a new compiler analysis and transformation framework for GPGPU programs implemented on LLVM, that can detect and meld similar control-flow regions in divergent paths to reduce divergence in control-flow. DARM generalizes and subsumes prior efforts at reducing divergence such as tail merging and branch fusion. We showed that DARM improves performance by improving ALU utilization and promoting coalesced shared memory accesses across several real-world benchmarks.

3. CFM-CS: CONTROL-FLOW MELDING FOR CODE SIZE REDUCTION

3.1 Motivation

Resource constrained environments such as embedded systems and mobile devices have limited memory and storage space, and thus, code size reduction becomes a critical compiler optimization for such systems [75]–[77]. Modern software systems are getting complex and large with various features and functionalities introduced regularly. Therefore, managing code size of such systems is a challenging but important task [78], [79].

Modern compilers have a variety of code size reduction techniques to reduce the code size of the generated executable [14], [80]. Classical compiler optimizations such as dead-code elimination [81], common sub-expression elimination [17], redundancy elimination [82], and constant propagation [83] focus on eliminating various redundant code patterns in the source program in order to reduce its size. Another popular approach for code size reduction is to detect identical or similar code sequences and merge them to remove the duplicates. Various function merging techniques have been proposed that are capable of merging identical functions [74], [84] or functions with similar instruction sequences [56], [85]. Most general versions of function merging rely on sequence alignment techniques to identify similar code sequences [55].

Despite the availability of various techniques that exploit code similarity to reduce code size, there are still many opportunities that are not being exploited by existing techniques. In this work, we identify one such opportunity, reducing code size by merging similar code within conditional branches (*i.e.* `if-the-else` constructs) in a program. Modern compilers like gcc [15] and LLVM [14] already contains optimizations that merge code sequences within conditional branches. Transformations like code sinking, code hoisting [13], and tail merging can merge identical instruction sequences contained in `if-then-else` branches by moving them to a common successor or predecessor blocks. One major limitation of these techniques is that they cannot fully exploit the code similarity within `if-then-else` branches because they only merge identical code sequences. If the code sequences within `if` and `else` branches are nearly identical with some differing instructions, these techniques cannot merge them.

More recently, Coutinho *et al.* [11] proposed *branch fusion* that can merge non-identical code sequences within diamond-shaped conditional branches. Branch fusion uses sequence alignment [54] techniques to identify similar instruction sequences within `if` and `else` paths of the branch and move them to common blocks. Unlike code sinking, code hoisting, and tail merging, branch fusion does not require the code sequences within `if` and `else` basic blocks to be identical. One common limitation of all these techniques is that they are only applicable at basic block level and cannot merge similar code at control-flow region level. For example, branch fusion is only applicable when the `if` and `else` paths of a conditional branch contain single basic blocks (*i.e.* diamond-shaped control-flow) [11]. Code sinking is applicable when all the predecessors of a block ends with unconditional branches and common code at the end of these predecessors can be *sinked* to the block [86].

```
1  #define DeleteNode(x)          \
2  { xx_hold = (x); \
3    while( Up(xx_hold)   != xx_hold ) \
4      DeleteLink( Up(xx_hold) ); \
5    while( Down(xx_hold) != xx_hold ) \
6      DeleteLink( Down(xx_hold) ); \
7    Dispose(xx_hold);          \
8  }
9  // ....
10 if( prnt_flush )
11 {
12   Parent(prnt, Up(dest_index));
13   if( kill )  DeleteNode(dest_index);
14   debug0(DGF, DD, "  calling FlushGalley ....");
15   FlushGalley(prnt);
16 }
17 else if( kill )  DeleteNode(dest_index)
18
```

Figure 3.1. Code snippet extracted from file `z20.c` in MiBench `typeset` benchmark

Region level code similarity is quite common in real-world programs. Code snippet shown in Figure 3.1 is an example of such missed opportunity. This code is extracted from function `ParenFlush` (source file `z20.c`) in MiBench `typeset` benchmark [87]. This function contains

an `if-then-else` branch with `if-then` branch inside the `if` and `else` paths (highlighted lines 13 and 17 in Figure 3.1). Both `if-then` statements inside the branch calls `DeleteNode` function with the same argument. `DeleteNode` function is a macro function (lines 1-8 in Figure 3.1) containing multiple while loops and several other macro function calls. Macro expansion causes the top-level `if-then-else` branch to contain large isomorphic control-flow regions with nearly identical instruction sequences within corresponding basic blocks of the two matching regions. Traditional techniques can not exploit this opportunity due to their inability to merge control-flow at region level and the fact that the code sequences within `if` and `else` paths are not identical.

Recently, Saumya *et al.* proposed Control-flow Melding (DARM) [88]. DARM is a compiler technique that merges control-flow regions within divergent `if-then-else` branches in GPU programs to reduce control-flow divergence. Even though DARM is capable of merging control-flow at region level it has several limitations that prevent it from being used for general purpose code size reduction. First, DARM only works on branch paths that contain simple control-flow regions, *e.g.* nested `if-then`, `if-then-else` or *natural loops*, that are isomorphic. Second, DARM is designed to optimize GPU programs and thus, it is not directly applicable to CPU programs. For example, DARM uses GPU latency cost to find the best possible alignment of instructions within matching isomorphic control-flow regions.

In this work, we propose CFM-CS, an extension of DARM that can exploit both control-flow structure similarity and instruction sequence similarity to reduce code size in real-world CPU programs. Unlike DARM, CFM-CS can handle complex control-flow regions within `if-then-else` branches and uses LLVM's built-in code size cost model [58] to decide the profitability of its applications. Since the LLVM's built-in cost model allows us to reason about the profitability of the transformation at compile time, we can only apply CFM-CS when it is profitable. In fact, out implementation of CFM-CS can reduce the LLVM-IR size of the `ParenFlush` function shown in Figure 3.1 by 28.8% (from code size cost of 236 to 168). We make the following contributions in this work:

- We propose CFM-CS, a novel code size reduction technique based on DARM that can merge control-flow at region level to reduce code size in CPU programs.

- An implementation of CFM-CS in LLVM that is publicly available[1].
- An evaluation of CFM-CS in 3 CPU benchmark suites showing its effectiveness in reducing code size in a variety of applications.

3.2 Extending Control-Flow Melding for Code Size Reduction

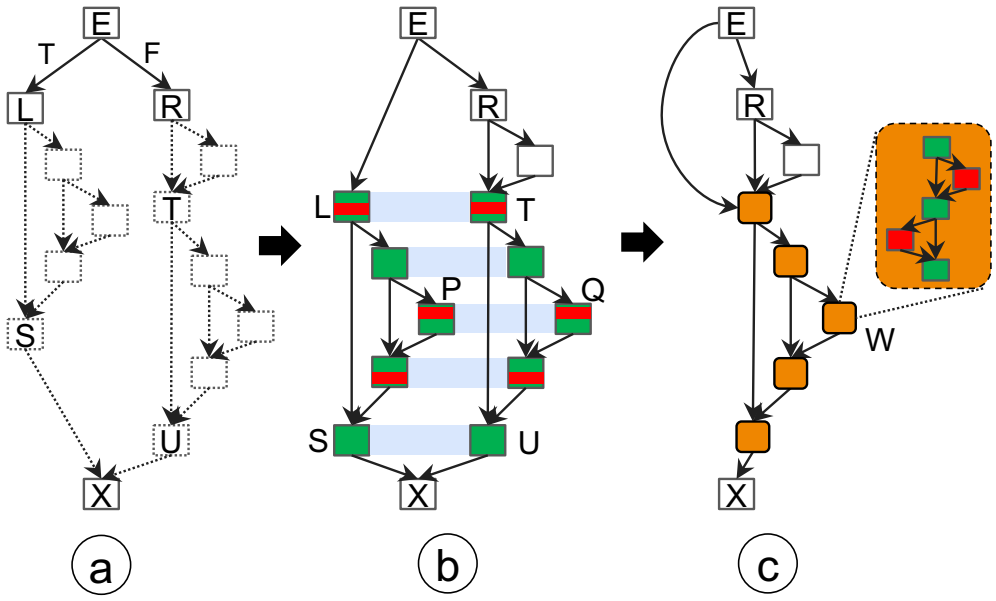

Figure 3.2. CFM-CS overview. (a) Given an *if-then-else* statement, (b) we identify isomorphic control-flow in the two regions, and (c) we align and merge the corresponding blocks.

Control-flow Melding [88] (DARM) is a code optimization technique used for reducing control-flow divergence in GPU programs. DARM reduces divergence by merging similar control-flow regions contained within divergent branches of the CFG. Previous compiler-based divergence reduction techniques such as Tail Merging and Branch Fusion are unable to merge control-flow beyond basic block boundaries. Therefore, they have limited applicability in real-world programs. DARM was proposed to fill this gap and enable merging control-flow at region level. DARM works by merging structurally similar (*i.e.* isomorphic) single-entry single-exit (SESE) regions within *if-then-else* branches. Even though the gen-

[1]↑https://github.com/charitha22/hybf-cc23-artifact/

eral idea of merging similar control-flow regions is applicable to real-world programs, DARM's implementation is fairly restrictive as it only supports merging simple nested *if/if-else* statements and *loops* inside *if-then-else* branches.

In this work, we extend and adapt DARM to reduce code size in CPU programs. In the following sections we describe the main steps in Control-flow Melding for Code Size Reduction (CFM-CS). Figure 3.2 shows the main stages of CFM-CS.

3.2.1 Identifying Regions for Melding

The first step of CFM-CS is identifying on which locations to apply the transformation. Similar to DARM, CFM-CS is also applicable to *if-then-else* constructs that contains isomorphic control-flow regions. To formally describe the conditions that a valid location must satisfy, consider the CFG in Figure 3.2 ⓐ. This CFG contains a basic block E with a conditional branch at its end. Basic blocks L and R be the two successors of E. Let X be the immediate post-dominator of E. E dominates all basic blocks contained within the SESE region E-X. E is considered to be a valid location for our transformation if there exist no paths in the CFG from E to X that goes through both L and R. This ensures that either L or R is executed at a time but not both, enabling us to *at least* merge the common computations within L and R. If there exists a path from L to R at least one predecessor of R must be dominated by L because all program paths from E to X must go through either L or R. We use this property to check non-existence of paths from L-R or R-L. In addition, basic blocks contained within E-X must not contain switch-case instructions for CFM-CS to be applicable. This is only a limitation of our current implementation, and if switch-case instructions can be converted to branches CFM-CS can still be applied.

The next step of CFM-CS is to collect all the subregions contained with the parent region of E-X. We employ LLVM's region tree (*i.e.* region hierarchy graph) [53] data structure to do this. We collect subregions along the left path (from L to X) and right path (from R to X). Each subregion is selected such that subregion entry is dominated by L or R and subregion exit post-dominates L or R. For example, the CFG in Figure 3.2 ⓐ has the subregion L-S on left path and subregions R-T,T-U on the right path. Any isomorphic SESE subregion

pair consisting of one subregion from left and right paths can be merged to potentially reduce code size. We use a heuristic-based approach based on instruction frequencies and their size cost to determine what isomorphic subregion pairs to merge. Isomorphic SESE subregions with more similar instructions are more profitable to be merged together. We formulate this as a sequence alignment problem and solve it using the Smith-Waterman algorithm [54]. For example, in Figure 3.2 ⓑ isomorphic subregions L-S and T-U are aligned together and their corresponding basic blocks (shown connected with light blue bars) can be merged.

3.2.2 CFM-CS Code Generation

We compute an instruction alignment similar to DARM [88] or HyFM [85], to generate the final merged regions. Instruction alignment is computed for each corresponding basic block pair in aligned subregions. In Figure 3.2 ⓑ portions of basic blocks with perfectly aligned instructions are shown in green and unaligned portions are shown in red. We use instruction alignment to generate the final merged code (shown in Figure 3.2 ⓒ). The aligned instructions are replaced with merged instructions that use *select* instructions to pick their operands, while the unaligned instructions are moved to new basic blocks and executed conditionally. For example, matched basic blocks P and Q have both aligned and unaligned portions and the final merged CFG for these blocks are shown zoomed-in on Figure 3.2 ⓒ. Note that the orange colored blocks are not necessarily basic blocks but control-flow subgraphs. We use the branching condition at block E as the distinguishing predicate for the *select* operations as well as for conditionally executing unaligned instructions.

3.2.3 Region Replication

CFM-CS is only capable of merging isomorphic regions, however there is one exception. When one path contains only a single basic block and the other path contains region(s), CFM-CS can still be applied by using *Region Replication*. The idea here is to replicate a region and place the single basic block in a convenient location to enable profitable merging. Region replication was first proposed in DARM [88] to merge basic blocks in *if-else-if* chains to reduce control-flow divergence. Their implementation did not support more complex control-

flow patterns because the primary focus was to reduce divergence in some select control-flow patterns seen in GPU kernels. We build on top of DARM and provide a more general region replication approach that is applicable to code size reduction in CPU programs. We use example CFGs shown in Figure 3.3 to explain how region replication works.

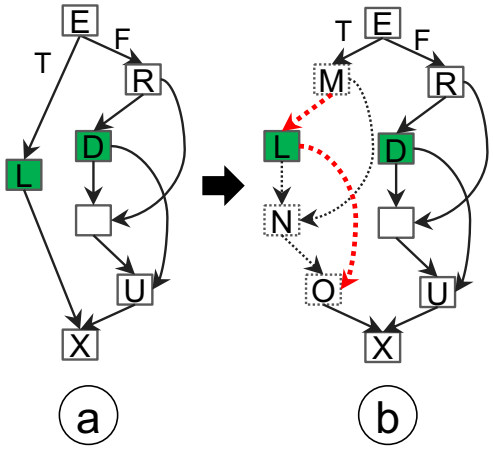

Figure 3.3. Region replication example

The input CFG (block L in Figure 3.3 ⓐ) contains a single basic block on the left path and a subregion in the right path. Assume that the computations done in blocks L and D are similar and merging them is profitable. First we replicate the right subregion R-U and create a new subregion M-O (Figure 3.3 ⓑ). And then we place L on a corresponding position to D. This creates two isomorphic regions and we can apply CFM-CS region merging approach[2]. We also make sure values produced in L will reach their external users by inserting ϕ nodes at L's new dominance frontiers (in this case L has two dominance frontiers N and O). We concretize the path conditions on region M-O to make sure L is always executed (concretized path $M \rightarrow L \rightarrow O$ is shown in red) and also make sure phi-nodes in block X pick the correct incoming values based on the chosen path.

[2]↑Alternatively, blocks L and D in Figure 3.3 ⓐ can be merged directly and direct jumps from $E \rightarrow D$ and $D \rightarrow X$ can be inserted to ensure correct control. CFM-CS does not use this approach because applying it recursively can make the CFG more complex/unstructured and unamenable to other optimizations including CFM-CS

3.3 Evaluation

In this section, we evaluate our implementation of CFM-CS on several CPU benchmark suites to measure its effectiveness in reducing code size.

3.3.1 Evaluation Setup

We implemented CFM-CS as a LLVM transformation pass[3]. We use LLVM's built-in target-specific code size cost model to estimate the benefit of CFM-CS transformation at compile-time. CFM-CS can be implemented on any static single assignment (SSA) [89] based intermediate representation and does not depend on any LLVM-IR specific feature to the best of our knowledge. For the baseline, we use `-Oz` optimization level in `clang` because the optimization pipeline in `-Oz` is designed for code size reduction. CFM-CS is applied on top of `-Oz` optimization level to mesure the improvement over the baseline. We place CFM-CS after the classic redundancy elimination and code motion passes in the pass pipeline, as they can be negatively affected by branch fusion. We evaluate CFM-CS with `-Oz` baseline on four different benchmark suites: AnghaBench [90], MiBench [91], SPEC 200, and SPEC 2017 [92]. These benchmarks cover a variety of applications including compilers, interpreters, typesetting, 3D rendering, and cryptography. We perform all experiments on a server with two octa-core Intel Xeon E5-2650 processors and 64 GiB of RAM, running Ubuntu 18.04.3 LTS. For timing measurements, we repeat all experiments 10 times to minimize the effect of measurement noise.

3.3.2 Code Size Reduction

We use `llvm-size` [93] to measure the size of the *text* section of the generated binaries. Table 3.1 shows the code size reduction achieved by CFM-CS on MiBench benchmarks. We show the absolute reduction of size in bytes, percentage reduction and the number of profitable applications of CFM-CS on each benchmark. The average absolute size reduction for MiBench is 333.8 bytes. `typeset` benchmark shows the largest absolute reduction of 2160

[3]↑LLVM-14.0

bytes. This benchmark has 40 profitable applications of CFM-CS, which is the second highest among all the benchmarks in MiBench. It contains multiple conditional branches with both straight line code complex control-flow regions where CFM-CS can be applied.

Table 3.1. Code size reduction achieved by CFM-CS on MiBench benchmarks

Benchmark	Reduction (Bytes)	Reduction (%)	Number of Fusions
blowfish	624	3.9	2
ghostscript	1232	0.1	64
gsm	0	0.0	1
ispell	32	0.1	4
jpeg_c	152	0.1	16
jpeg_d	264	0.2	21
patricia	8	0.2	2
pgp	128	0.1	12
rsynth	48	0.1	4
susan	0	0.0	0
tiff2bw	88	0.0	8
tiff2dither	88	0.0	8
tiff2median	88	0.0	8
tiff2rgba	88	0.0	8
typeset	2160	0.4	40

Our motivating example (`ParentFlush` function) in Section 3.1 is also from this benchmark. Application of CFM-CS on `ParentFlush` function reduces its LLVM-IR size by 28.8%. Other most profitable applications of CFM-CS on this benchmark are in `FilterFlush` (13.79 % reduction), `TransferEnd` (13.9 % reduction), and `Meld` (3.17% reduction) functions. None of the benchmarks in MiBench have code size increases which shows that the LLVM cost model is suffciently accurate to estimate the benefit of CFM-CS transformation. However, the percentage reduction in code size is not very high. The highest percentage reduction was observed for `blowfish` benchmark (3.9%). In fact, none of the other benchmarks have more than 1% reduction in code size. This is mainly because the number of opportunities for CFM-CS is low relative to the size of the benchmark. Also, the reported percentage reduction is for the final binary of the benchmark which includes large number of functions without any opportunities for CFM-CS. We observe that absolute size reduction is higher when there

are more opportunities for CFM-CS. This can be observed in `typeset` and `ghostscript` benchmarks where the absolute reduction is 2160 and 1232 bytes respectively and CFM-CS is applied 40 and 64 times respectively.

Table 3.2. Code size reduction achieved by CFM-CS on SPEC 2006 benchmarks

Benchmark	Reduction (Bytes)	Reduction (%)	Number of Fusions
400.perlbench	304	0.0	4866
401.bzip2	64	0.1	4
403.gcc	1712	0.1	115
433.milc	80	0.1	6
445.gobmk	296	0.0	65
447.dealII	1080	0.0	57
450.soplex	704	0.2	74
453.povray	96	0.0	45
456.hmmer	64	0.0	14
458.sjeng	200	0.2	31
462.libquantum	72	0.2	10
464.h264ref	2296	0.5	93
471.omnetpp	-24	-0.0	4
473.astar	-80	-0.2	5
482.sphinx3	152	0.1	8
483.xalancbmk	324	0.0	60

Tables 3.2 and 3.3 show the results for SPEC 2006 and SPEC 2017 benchmarks respectively. The general observation is quite similar to MiBench benchmarks. In SPEC 2006 benchmarks, the largest absolute size reduction was observed in `602.gcc_s` benchmark (6760 bytes). CFM-CS is applied 482 times in this benchmark. There are 3 benchmarks where the size reduction is more than 1 KB (`403.gcc`, `dealII`, and `h264ref`). Largest absolute reduction is in `h264ref` benchmark (2296 bytes) and CFM-CS applies to 93 locations in this benchmark. However, similar to MiBench, the percentage reduction is not very high. Most benchmarks in SPEC suites contains very larges number of functions and there are many functions that CFM-CS does not apply to. Therefore, the percentage reduction of the final binary size is not low. In SEPC 2006 the highest percentage reduction was observed in `464.h264ref` benchmark (0.5%). In SPEC 2017, the largest absolute size reduction is observed in `526.blender_r` benchmark (8672 bytes) where CFM-CS is applied 570 times.

There are 3 benchmarks where the size reduction is more than 1 KB (`526.blender_r`, `600.perlbench_s`, and `602.gcc_s`). The percentage size reduction is less than 1% in all benchmarks, with `631.deepsjeng_s` having the highest percentage reduction (0.8%).

Table 3.3. Code size reduction achieved by CFM-CS on SPEC 2017 benchmarks

Benchmark	Reduction (Bytes)	Reduction (%)	Number of Fusions
511.povray_r	64	0.0	48
526.blender_r	8672	0.1	570
600.perlbench_s	1592	0.1	146
602.gcc_s	6760	0.1	482
605.mcf_s	112	1	0.6
620.omnetpp_s	72	0.0	18
625.x264_s	984	0.2	33
631.deepsjeng_s	592	0.8	25
638.imagick_s	-200	-0.0	118
641.leela_s	72	0.1	10
644.nab_s	216	0.1	10
657.xz_s	-32	-0.0	3

In SPEC 2006 and SPEC 2017 benchmarks, there are several cases where CFM-CS causes an increase in the binary size. In SPEC 2006 benchmarks, `471.omnetpp` and `473.astar` benchmarks have a negative size reduction (increase of 24 and 80 bytes respectively). And in SPEC 2017 benchmarks, `638.imagick_s` and `657.xz_s` benchmarks have a negative size reduction (increase of 200 and 32 bytes respectively). This is caused by the inaccuracy of the compile-time probability analysis. Reasoning about the binary codes size at LLVM-IR level is not entirely accurate. For example, in LLVM-IR level code size cost of ϕ instructions is zero because they can be converted to simple copies or completely eliminated in the backend code generation phase (register allocation). However, this estimate is not entirely accurate because some cases ϕ instructions can get translated into more expensive operations in later passes. The code size increase was observed in benchmarks where CFM-CS applies to smaller number of locations and the size of the merged regions are small (*i.e.* small conditional branches with limited amount of similar code). In such cases, CFM-CS ends up adding more overhead (ϕ instructions) than the size reduction achieved by merging the code. And the

overhead is not accurately captured by the compile-time probability analysis. In summary, we can say that CFM-CS is capable of reducing binary size of large benchmarks. Even though the percentage reduction is not very high, the size reduction achieved by CFM-CS can still be significant given that the compile-time and runtime overhead introduced by CFM-CS is not very significant. In resource constrained environments, such as embedded systems, CFM-CS can still be a useful optimization.

3.3.3 Code Size Reduction on Individual Functions

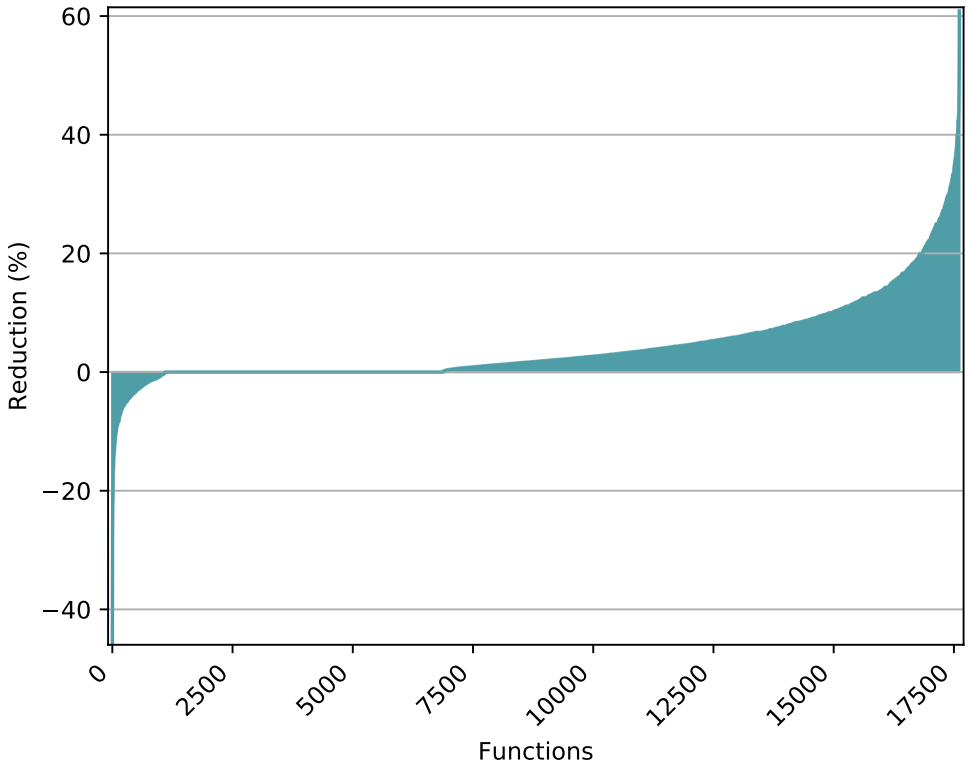

Figure 3.4. Reduction in number of instructions on 17.6k real world-functions from AnghaBench suite. Functions are sorted by the amount of reduction achieved by CFM-CS.

To investigate the effectiveness on real world functions, we applied CFM-SE to 17.6k functions from AnghaBench [90] suite. AnghaBench suite consists of one million independently compilable functions extracted from popular GitHub repositories containing `C` source files.

We measured the reduction of LLVM-IR instructions in each function. The results are shown in Figure 3.4. In this figure, the functions are sorted by the amount of reduction achieved by CFM-CS (left side least reduction in instructions to right side most reduction in instructions). CFM-CS can reduce the number of instructions in substantial portion of the functions considered obtaining more than 60% reduction in some cases.

There are cases where the number of instructions is increased by CFM-CS. These regressions are caused by the ϕ instructions inserted by CFM-CS during the merging process. In LLVM cost model ϕ instructions have zero code size cost and therefore profitability analysis does not control the number of ϕ instructions inserted by CFM-CS. If the number of instructions merged by CFM-CS is low relative to the number of ϕ instructions and `select` instructions inserted, then the number of instructions can increase. This can be observed in functions with many small `if-the-else` branches with limited amount of similar code. Even though ϕ instructions are considered cheap in terms of code size they can get translated into more expensive operations later in the compilation pipeline. Some of binary code size increases described in Section 3.3.2 are also caused by the limited accuracy of the compile-time probability analysis. This issue can be mitigated by setting a threshold on the amount of code size reduction that must be achieved by CFM-CS before it is applied to a function. In our implementation, we do not use such threshold and CFM-CS is applied whenever there is a positive code size reduction.

3.3.4 Compile-Time and Runtime Overhead

We also measured the compile-time and runtime overhead of CFM-CS for the all the benchmark suites. Average compile time overhead across all benchmarks is 5.5%. The highest compile time overhead was observed for the `blowfish` benchmark where compile time increased by 29%. `blowfish` benchmark contains an `if-then-else` branch with each diverging basic block containing 403 LLVM-IR instructions. Therefore, CFM-CS spends significant time in the instruction alignment computation which has quadratic complexity in the number of instructions in the basic block [54]. `blowfish` also has good reductions in code size as well, therefore the compile time overhead invested in CFM-CS is not wasted in this case. We

also note that our implementation of CFM-CS is not optimized for compile time. Compile time can be improved by using better sequence alignment algorithms and by using more efficient data structures in the implementation. For example, rather than using an alignment algorithm with quadratic complexity, we can use a less accurate but faster algorithm for the instruction alignment computation. For measuring the runtime overhead we considered 30 benchmarks from MiBench, SPEC 2006 and SPEC 2017 that are modified by CFM-CS transformation. We observed an average runtime *improvement* of 2.3% across all benchmarks. However, this average is heavily influenced by the `tiff2bw` benchmark where the runtime improvement was more than 50%. Several other benchmarks also showed improvements in runtime performance (`blowfish_e`, `ghostscript`, `tiff2median`) even though they were not significant.

3.4 Related Work

Compiler based code size reduction is an important optimization that enables fitting larger applications into resource constrained environments like mobile phones or embedded systems [75], [77], [79]. Well known code size reduction techniques include replacing a fragment of code with a smaller semantically equivalent code sequence [94], combining redundant code within function or across functions [17], [95], eliminating redundant code [16].

Production compilers like GCC and LLVM [74], [96] provide an optimization for merging identical functions at the IR level. Type mismatches are allowed but only if the types can be losslessly casted to same format. Von Koch et al. [84] extended this to merging nearly identical functions. This approach can only merge functions with same signature and identical control-flow graphs. In addition, matching basic blocks must have the same number of instructions and paired instructions within corresponding blocks must have same data types.

Rocha et al. [55], [56] proposed an approach for merging arbitrary pair of functions. The use sequence alignment techniques to find out common instruction subsequences within two functions. Common sequences gets merged together while unaligned sections of the two functions are conditionally executed using a function identifier. Due to the sequence

alignment computations, this approach has high compile time overhead. HyFM [85], [97] attempts to accelerate arbitrary function merging by restricting the alignment computation to basic block level and using a simple linear alignment strategy. HyFM is much faster than its predecessor.

Chen *et al.* proposed *Generalized Tail Merging* for code size reduction [98]. This extends the tail merging to work on isomorphic SEME regions. This approach is conceptually similar to CFM-CS, but it requires matching basic blocks to contain nearly identical instruction sequences. This restricts the applicability of generalized tail merging to a subset of cases that CFM-CS can handle.

3.5 Conclusion

Existing techniques that exploit code similarity in conditional branches to reduce code size only applicable in limited scenarios. These techniques work only when the code sequences contained in the divergent paths of a branch are identical. If the instruction sequences are not identical but have high similarity, they fail to capitalize on the full opportunity. Also existing techniques does not fully exploit the structural similarity of the control-flow graph to reduce code size. In this work, we propose CFM-CS, an extension of control-flow melding designed for code size reduction in CPU programs. CFM-CS uses a hierarchical sequence alignment approach to exploit both structural and instruction similarity to reduce code size. CFM-CS approach is more general and can handle wide variety of cases for code size reduction in conditional branches and more general than existing techniques like code hoisting/sinking and tail merging. Our evaluation of CFM-CS shows that it is applicable in variety of well-know CPU benchmarks and achieves decent code size reduction without significant performance or compile-time overhead.

4. CFM-SE: ACCELERATING SYMBOLIC EXECUTION BY TARGETED CONTROL-FLOW TRANSFORMATIONS

4.1 Introduction

Computer software govern almost every aspect of human life. Therefore, ensuring software performs its task according to the given specification has paramount importance. Verification and testing of computer software play a vital role in ensuring their correct operation. *Dynamic Symbolic Execution (DSE)* is popular dynamic analysis technique used for software testing and verification [99], [100]. DSE executes a program using symbolic variables instead of concrete values as input. With some variables declared as symbolic, DSE can explore all feasible paths in a program. A program path is *feasible* if there exists at least one input that will exercise that path. For each explored path DSE computes a *path condition* which is essentially the conjunction of branching conditions that are true along the path. When DSE reach a branch where the branching condition is symbolic, it continues the execution on both directions of the branch (*true* and *false*) if they are feasible. This path condition can be solved using an SMT solver [101], [102] to find a concrete input which exercises that path. Because DSE explore all feasible paths in a program, it can be used to find bugs or prove the program works correctly for all possible inputs.

Unfortunately, DSE suffers from the *path explosion problem*, wherein the number of paths grows exponentially with the number of symbolic branches in the program [20], [103]. Complex control-flow (*i.e.* code with a lot of branches) is the main contributing factor for path explosion. For example, consider a program with a symbolic branch inside a loop. For each loop iteration there are two potential paths through the program that DSE needs to explore. If the loop has N iterations number of paths amounts to 2^N. Out of the many techniques [19], [27]–[29], [104], [105] proposed for mitigating path explosion problem *dynamic state merging* [26] is considered one of the foundational techniques. In dynamic symbolic execution engines like KLEE [100], each explored path is associated with a *state* which maintains the values of all symbolic variables, memory, stack and registers at that point in the program. Often times multiple paths in a program share the same state or very similar states. State merging exploits this observation by merging sufficiently similar

states together to reduce the number of paths that need to be explored. State merging uses both dynamic and static program information to figure out which states are profitable to merge. Even though state merging can reduce the number of paths that need to be explored significantly, it still requires calling the SMT solver at symbolic branches. At conditional branches where both the *true* and *false* are feasible, calling the SMT solver to check the path feasibility is an unnecessary overhead in cases where the two forked states get merged.

Prior work suggests that static program transformations can also be used to improve the performance of dynamic test generation [30], [106]. In this context of DSE, both semantics-preserving [107] and non-semantics-preserving program transformations [108] have been proposed to improve its performance. The root cause of path explosion is the symbolic branches in a program. If the number of symbolic branches can be reduced, the number of paths that need to be explored will also be reduced. For example, Collingbourne *et al.* [33] used aggressive *phi-node folding* [109] to reduce the number of symbolic branches in image processing applications and showed that it can greatly improve the performance of DSE on programs operating on images. Well-known compiler optimizations like code hoisting/sinking or tail merging can be used to reduce the number of symbolic branches in a program. Tail merging can completely eliminate `if-then-else` branches if the `then` and `else` paths contain identical operation sequences. This is done by merging instruction with identical opcodes with the help of `select` instructions. This approach can merge diamond shaped control-flow patterns (*i.e.* `if-then-else` branches) into a single basic block. LLVM [14] optimizer contains control-flow graph simplification pass (`-simplifycfg`) that can eliminate branches by hoisting or sinking instructions out of `if-then-else` or `if-then` statements when the compiler can prove it is safe to do so. In KLEE, the `-simplifycfg` pass is enabled by default to perform these branch elimination optimizations.

Branch elimination optimizations in modern compilers are designed to improve the performance of the generated code on various hardware platforms. These optimizations are unaware of performance implications of these transformations on DSE. For example LLVM `simplifycfg` pass uses target specific cost models to determine if it is profitable hoist/sink common instructions out of `if-then-else` or speculatively execute `then` block of an `if-then` statement [14]. Often times these transformations does not apply to program locations

that can improve the performance of DSE, or they might be applied to locations that can degrade or have no impact on DSE performance (e.g. concrete branches). Therefore, a principled approach is needed to identify and apply branch elimination optimizations that can improve the performance of DSE.

Recent developments in compilers like DARM [88] and HyBF [110] have shown how to exploit code similarity within conditional branches to improve performance and code size of generated code [88], [110]. These approaches work by moving common instruction subsequences out of conditional branches and into a separate basic block. Even though this generalizes hoisting/sinking optimizations, they increase the number of branches in the program and select instructions in the generated code if applied to conditional branches with non-identical instruction sequences. This can hurt the performance of DSE because it can increase the number of symbolic branches and increase the complexity of the path constraints due to the additional `select` instructions inserted.

In this paper, we propose CFM-SE, a targeted control-flow transformation that is designed to remove expensive symbolic branches from a program to improve the performance of DSE. First, CFM-SE uses static analysis to identify symbolic branches that are expensive to explore in DSE. Then it uses DARM [88] framework to identify code similarity within conditional branches. Next CFM-SE inserts minimal additional *dead* instructions to `if` and `then` blocks (possibly empty `if-then` statements) to make them look identical in terms of operation sequences. Finally, CFM-SE merges the identical instruction sequence within the `if` and `then` blocks into a single basic block to eliminate the expensive symbolic branch.

DARM transformation is not semantics-preserving because unconditional execution of certain instructions (*e.g.* load/stores) is not safe. This can introduce new bugs to the program that were not present in the untransformed program. However, DARM is *failure-preserving*. A failure preserving transformation ensures that any bug present in the untransformed program is also present in the transformed program. DARM is failure-preserving because the additional instructions inserted into the program does not alter the original computation or program memory state. Any crashing input resulted after a failure-preserving transformation can be checked against the original program to verify whether the crash is a true-positive or not.

We use this property to develop a framework to detect false-positive bugs introduced by failure-preserving transformations like DARM that is not semantic-preserving.

The main contributions of this paper are as follows:

- We propose CFM-SE, a targeted non-semantics-preserving and failure preserving control-flow transformation that is designed to remove expensive symbolic branches from a program to improve the performance of DSE.
- An implementation of CFM-SE in LLVM.
- A framework for detecting false positives caused by non-semantics-preserving transformations like CFM-SE in the context of DSE.
- Evaluation of CFM-SE, showing its ability to improve the performance of DSE on a variety of benchmarks.

4.2 Background

4.2.1 Dynamic Symbolic Execution and State Merging

Dynamic Symbolic Execution (DSE) is a dynamic program analysis technique that can enumerate all feasible execution paths of a program. DSE executes a program using symbolic inputs and uses an SMT solver [101], [102] to reason about feasibility of the execution path at branch points in the program. DSE suffers from path explosion problem where the number of feasible execution paths can grow exponentially with the number of branch points in the program. State merging [26] mitigate the path explosion problem by merging sufficiently similar program paths during DSE. Even though highly effective at reducing the path explosion, state merging still need to call the SMT solver at every branch point of the program.

4.2.2 Divergence Analysis

Divergence analysis is used for identifying divergent variables in a program. Identifying divergent instructions is crucial for some GPU-specific compiler optimizations. For example,

branch fusion [11] and DARM [88] uses divergence analysis to identifying which control-flow regions to merge to reduce control-flow divergence. In GPU compilers, a branch is marked as divergent if the branch outcome can be different for threads in a thread group (*i.e.* warp). Some program variables such as thread IDs or global memory reads are by definition divergent. Such variables are called *divergence sources*. LLVM divergence analysis is an intra-procedural data-flow analysis. Every instruction in a function is marked as *divergent* if it is *data-dependent* or *sync-dependent* on a divergence source or another divergent instruction. Here, sync-dependance captures the control-flow aspect of divergence. In a branch such as `if (a) {b = 1;} else {b = 2;}`, variable `b` is assigned concrete values in both `true` and `false` paths but, `b`'s users can be divergent if the branching condition `a` is divergent. This is because the value of `b` can be different based on the outcome of the divergent branch.

4.2.3 DARM

DARM [88], [110] (*i.e.* Control-flow Melding) is a compiler optimization that exploits code similarity at control-flow region level to improve code size and performance. DARM employs a hierarchical sequence alignment technique to identify isomorphic control-flow regions that contains similar instruction sequences within them. If two isomorphic regions are similar enough (according to a cost model [58]), DARM merges them into a single region. By changing the alignment models in DARM can be applied to different applications such improving performance in GPU applications or reducing code size in CPU applications. DARM provides a flexible way to exploit code similarity at control-flow region level and, it is more general than traditional compiler optimizations such as code sinking/hoisting, tail merging [13] or branch fusion [11] that exploits code similarity only at the basic block level.

4.3 Motivating Example

Our motivating example is `to_upper` function that coverts all elements of a `char` array to upper case and, it is shown in Figure 4.1. This figure also shows the driver in `main` method to symbolically execute `to_upper` function with a char array of size `SIZE` as input. After

the execution of `to_upper`, the driver also asserts that the output array contains no lower case characters. In this case, scalability of this example is limited, since symbolic execution engine has to fork the execution at every iteration on the branching condition inside the loop (line 3) which is symbolic. In fact, when this program is run with KLEE[1] using default settings, it explores 1024 program paths and invokes the SMT solver 21 times for input size 10. If constraint caching is disabled, number of SMT solver invocations increases to 90.

```
1  void to_upper(char *text) {
2    for (int i = 0; i < SIZE; i++) {
3      if ((text[i] >= 'a') & (text[i] <= 'z'))
4        text[i] = text[i] - 'a' + 'A';
5    }
6  }
7  void to_upper_branchless(char *text) {
8    for (int i = 0; i < SIZE; i++) {
9      unsigned is_lower
10       = (text[i] >= 'a') & (text[i] <= 'z');
11     unsigned diff = is_lower == 0 ? 0 : 'a' - 'A';
12     text[i] = text[i] - diff;
13   }
14 }
15 int main() {
16   char text[SIZE];
17   klee_make_symbolic(&text, sizeof(text), "text");
18   to_upper(text);
19   for (int i = 0; i < SIZE; i++){
20     klee_assert(
21       !((text[i] >= 'a') & (text[i] <= 'z')));}
22   return 0;
23 }
```

Figure 4.1. `to_upper` function and its branchless implementation (`to_upper_branchless`) with driver code for symbolic execution and verification of final result using asserts.

The conditional branch inside the loop can be removed by converting the control-flow into data-flow. The transformed function `to_upper_branchless` is also shown in Figure 4.1. Idea here is to compute how each character value must be shifted to convert it to upper case. If the character is lower case then the shift value is `'a' - 'A'` otherwise it is zero. We compute this value (*i.e.* `diff`) conditioned (line 11) on the character being lower case

[1]↑KLEE-2.3+LLVM-14.0

(*i.e.* `is_lower`) (line 9,10). And then we apply the shift to the character value (line 12). Note that the conditional assignment to `diff` is translated into a `select` instruction in LLVM-IR. KLEE converts `select` instructions into `ite` expressions therefore executing the loop does not require SMT solver invocations. If we run the transformed version in KLEE, it explores only one program path and invokes the SMT solver only 11 times (20 with constraint caching disabled). This example shows the utility of converting control-flow into data-flow in the context of DSE. Loops with symbolic conditionals are a common source of scalability issues in DSE. Targeted compiler transformations can be used to remove such bottlenecks.

Converting control-flow into data-flow is not safe when computations with side-effects are present inside the branch. In `to_upper` function, store to `text[i]` is an operation with side-effect because it modifies the input array. Therefore, compiler cannot hoist the store outside the conditional branch. In transformed code, store is executed unconditionally but, the stored value is the same as the original value if the character is not lower case. Even though this transformation is safe in this example, reasoning about its safety at compile time is not always trivial. But we argue that such branch eliminating transformations are useful in managing the scalability issues of DSE. Applying such transformations to the program can change the semantics of the program but might lead to better scalability of DSE. This can help in identifying bugs faster and increase coverage of DSE within limited amount of time.

In the next sections, we describe a compiler transformation called DARM that converts control-flow into data-flow to eliminate branches in a program to improve the scalability of DSE. DARM is non-semantics-preserving and may introduce new bugs in the program that were not present in the original program. Next we describe a system that allows us to filter out the false positives and verify if the bug discovered after DARM transformation is indeed a real bug in the original program.

4.4 Detailed Design

In this section, we describe the algorithm used by CFM-SE to statically merge paths in a program in order to accelerate DSE of the target program. We describe the dead code

insertion phase used for making the computation sequences within *if-then-else* statements identical, and the code generation phase used for eliminating branches that are expensive for DSE to explore. We describe the technique used by CFM-SE to identify symbolic values in the program using data-flow analysis. Finally, we describe a framework for filtering out false positive bugs that may be introduced by CFM-SE in to the target program.

4.4.1 CFM-SE Transformation

```
1  // ...
2  if ((text[i] >= 'a')
3    & (text[i] <= 'z')) {
4    t1 = text[i] - 'a';
5    t2 = t1 + 'A';
6    t3 = text[i];
7    text[i] = t2;}
8  else {}
```

(a)

```
1  // ...
2  if ((text[i] >= 'a')
3    & (text[i] <= 'z')) {
4    t1 = text[i];
5    t2 = t1 - 'a';
6    t3 = t2 + 'A';
7    t4 = text[i];
8    text[i] = t3;}
9  else {
10   t5 = text[i];
11   t6 = t5 - 0;
12   t7 = t6 + 0;
13   t8 = text[i];
14   text[i] = t8;}
```

(b)

```
1   // ...
2   unsigned is_lower =
3     (text[i] >= 'a') & (text[i] <= 'z');
4   t1_t5 =  text[i];
5   s1    = is_lower == 0 ? 0 : 'a'; // select
6   t2_t6 = t1_t5 - s1;
7   s2    = is_lower == 0 ? 0 : 'A'; // select
8   t3_t7 = t2_t6 + s2;
9   t4_t8 = text[i];
10  s3    =
11    is_lower == 0 ? t4_t8 : t3_t7; // select
12  text[i] = s3;
```

(c)

Figure 4.2. CFM-SE transformation example

CFM-SE transformation is based on Control-flow Melding (CFM) [88]. As we discussed in Section 4.2, CFM is a compiler optimization that improves performance of GPU programs by statically merging divergent program paths. Goal of CFM is to identify `if-then-else` branches with similar basic blocks (or isomorphic control-flow regions) and merge the common instructions within those blocks into convergent blocks. If the operation sequence inside both sides of the branch is identical, then the branch can be eliminated. However, this scenario is rare in real-world programs. If non-identical operations are found by the instruction alignment step, CFM moves the non-identical portions of the operations into new basic blocks and allowing them to execute conditionally. This process can increase the number of branches in the program. Therefore, CFM alone is not sufficient to be used as an optimization for improving DSE performance.

Key idea of CFM-SE is to insert dead instructions into the two sides of the conditional branch to make the operations sequences identical. Dead instruction is needed when the instruction alignment contains *unaligned* instructions. For the following definitions consider a program with an `if-then-else` branch with two basic blocks B_t and B_f (*i.e.* diamond shaped control-flow).

Definition 4.4.1. ***Instruction Alignment:*** *Let* $I_t = \{\mathrm{i}_1^t, \ldots, \mathrm{i}_n^t\}$ *and* $I_f = \{\mathrm{i}_1^f, \ldots, \mathrm{i}_m^f\}$ *be the ordered sequence of instructions in* B_t *and* B_f *respectively. An instruction alignment is an ordered sequence of item pairs* $A = \{(a_1, b_1), \ldots, (a_k, b_k)\}$ *such that* $a_\mathrm{i} \in I_t \cup \emptyset$, $b_\mathrm{i} \in I_f \cup \emptyset$, $k = \max(n, m)$ *and,* $(a_\mathrm{i}, b_\mathrm{i}) \neq (\emptyset, \emptyset)$ *for all* $\mathrm{i} \in [1, k]$. *If* $a_\mathrm{i} \neq \emptyset$ *and* $b_\mathrm{i} \neq \emptyset$, *then* a_i *and* b_i *are compatible for merging (i.e.* a_i *and* b_i *can be merged into a single instruction). Instructions are compatible if their operations and date types match.*

Definition 4.4.2. ***Unaligned Instruction:*** *Let* $(a_\mathrm{i}, b_\mathrm{i}) \in A$ *be an item pair in an instruction alignment* A *such that at least one of* a_i *and* b_i *is* $\emptyset$. *Let* i' *be the valid instruction out of* a_i *and* b_i. *Then,* i' *is called an unaligned instruction.*

Definition 4.4.3. ***Complete Alignment:*** *An instruction alignment* A' *is called complete if it does not contain any unaligned instructions.*

If the instruction alignment for B_t and B_f is *complete*, we can fully merge B_t and B_f into a single basic block eliminating the conditional branch. The first step of CFM-SE is to

transform the alignment A into a complete alignment A' such that A' becomes a complete alignment. Assume that i' is an unaligned instruction such that $\mathrm{i}' \in I_t$. We insert a dead instruction i'' into B_f such that in the new alignment A', i'' is aligned with i'.

Definition 4.4.4. ***Dead Instruction:*** *Assume that B_t contains an unaligned instruction. A dead instruction i'' is an instruction inserted into B_f or B_f such that i'' and i' are compatible and forms an item pair in the new alignment A'.*

Inserting dead instructions is necessary to make the instruction alignment complete that allows us to eliminate the conditional branch. But, inserting dead instructions can change the semantics of the program and can introduce new bugs. However, CFM-SE transformation ensures the following conditions to minimize the number of new bugs introduced by the transformation.

1. Dead instruction i'' is not used by any non-dead instruction in the program.
2. Operation of i' (or i'') can only be a side-effect free ALU operation or a memory read/write operation.
3. A dead memory operation is allowed to read from any memory location, but it is not allowed to change existing values in the memory.

Condition ① ensures that any of the original instructions in the program does not use any values produced by a dead instruction. Value flow in the original program is still preserved after inserting a dead instruction. Condition ② states that dead code insertion is not supported for all types of instructions (*i.e.* all opcodes). For example, if the unaligned instruction is a function call we cannot insert a dead function call because the function call can have side-effects. In CFM-SE we only insert dead instructions for ALU operations and memory read/write operations. Supported ALU operations include arithmetic operations, logical operations, comparison operations, bitwise operations, and conversion (*i.e.* casting operations) [111]. Condition ③ allows us to unconditionally execute load/store operations.

Select Minimization: In code generation process of CFM, extra select operation are inserted if the operands of the two merged instructions do not match. This process can increase

the number of instructions in the program and extra select operation can make the data flow more complex. In DSE, select operations essentially translate to `ite` expressions. More select instructions means more interpretation overhead and more complex constraints for the solver. Therefore, it is important to minimize the number of select instructions generated in the CFM-SE transformation. Select operations can be minimized if the both sides of the conditional branch have similar def-use chains. More precisely, let $\mathrm{i}_t = op(o_t^1, o_t^2)$ and $\mathrm{i}_f = op(o_f^1, o_f^2)$ be two aligned binary instructions in instruction alignment A. Merging i_t and i_f does not require additional select operations if $o_t^1 = o_f^1$ and $o_t^2 = o_f^2$ or (o_t^1, o_f^1) and (o_t^2, o_f^2) are also aligned instructions in the A.

Setting Operands for Dead ALU Instructions: There is some flexibility in setting operands for dead ALU instructions. On one hand, we can set all the operands of the dead instruction to some safe constant value (*e.g.* 0) depending on the semantics of the instruction in order to ensure safe execution of the dead instruction. On the other hand, we can try to preserve the def-use chains and minimize select operations. In CFM-SE, we do a mix of both approaches. We try to preserve the def-use chains and minimize select operations as long as the dead instruction can not result in any new bugs (such as overflow, underflow, division by zero or undefined behavior). The operand setting process is explained with an example at the end of this section.

Challenges in Merging Memory Operations: DSE engines such as KLEE experiences significant performance overhead if the program contains memory accesses to symbolic addresses [107]. Merging memory operations can result in symbolic memory accesses even if the two aligned memory operations access concrete addresses individually. For example, consider two load aligned load instructions $\mathrm{i}_t = load(a)$ and $\mathrm{i}_f = load(b)$ with a *symbolic* branching condition c and $a \neq b$. If we merge them into a single load the resulting load will be $\mathrm{i} = load(select(c, a, b))$. Even if a and b are concrete addresses, the resulting load will be symbolic because the address is a function of the branching condition.

Setting Operands for Dead Load/Store Instructions: To avoid creating more symbolic memory accesses, We follow the following criteria in aligning memory operations.

1. If the two aligned memory operations access the same address, we merge them into a single memory operation. If the addresses are same the merged instruction can not have a symbolic address.

2. If two aligned memory operations access concrete memory locations but, they are not the same, we convert them to unaligned memory operations. This essentially linearize the two memory operations but avoids creating symbolic memory addresses.

3. If at least one of the two aligned memory operations access a symbolic address, we don't apply CFM-SE to the branch.

Note that in case ②, linearizing guarded load/store instructions is not safe and can lead to new program bugs. This is because guarding condition might be protecting the memory operation against out-of-bounds access. Therefore, CFM-SE transformation can result in new out-of-bounds memory access bugs. We describe an automated way to filter out such false positive bugs in Section 4.4.3. If the memory location is valid, then CFM-SE transformation does not change the semantics of the program. This is because, the loaded value from a dead load is never used by any other instruction. For a dead store, the stored value is the same as the existing value in that memory location. This can be achieved by inserting a load from the same memory location before the dead store.

Algorithmic description of CFM-SE transformation is shown in Algorithm 3. This algorithm shows how CFM-SE transforms a given function F inside a module (consisting of many functions). The first step is converting all branches in F into a canonical form. In this step we convert `if-then` branches with a single basic block into diamond shaped `if-then-else` branches. This is step is important because control-flow melding implementation (*i.e.* DARM) requires this form. Next, we iterate through all the conditional branches in F collect all the branches that CFM-SE might be applicable. We filter out cases where the successor blocks of the branch contains symbolic addresses (Section 4.4.1) or contains program locations that CFM-SE is not allowed to merge (Section 4.4.3). Next, for each collected branch location we insert dead instructions to make the instruction sequences identical inside its left and right successors. We use DARM's instruction alignment mechanism to figure out the type of dead instruction that must be inserted and their location (Section 4.4.1). Then, the two successor

blocks of the branch are merged using control-flow melding. Note that merging changes the control-flow graph and introduce redundant unconditional jumps. These branches can reduce the number of opportunities for recursive applications of CFM-SE. Therefore, we remove the redundant jumps as the final step of the procedure. This while procedure is applied to each function of the module repeatedly until no more merging is possible.

Algorithm 3: CFM-SE Algorithm

```
Input: Function F, Location constraints LC
Output: Boolean changed indicating if F was modified or not
List branchL ← ∅
changed ← false
ConvertIfThenBranches(F)
for each block B in F do
    if B does not end with a conditional branch then
        continue
    end
    BI ← getTerminator(B)
    SuccL, SuccR ← getSuccessors(BI)
    if SuccL, SuccR has diamond-shaped control-flow then
        if SuccL or SuccR contains symbolic addresses then
            continue
        end
        if SuccL or SuccR does not satisfy location constraints LC then
            continue
        end
        branchL.append(BI)
    end
    for each branch instruction BI in branchL do
        SuccL, SuccR ← getSuccessors(BI)
        InsertDeadCode(SuccL, SuccR)
        if merge(SuccL, SuccR) then
            changed ← true
        end
    end
    if changed then
        RemoveRedundantJumps(F)
    end
    return changed
end
```

Example: Now we explain how CFM-SE transformation works in action using our running example (Figure 4.1). Figure 4.2 shows how `to_upper` function is transformed on each stage. Figure 4.2a shows `to_upper` function with empty `else` section inserted. This is extra canonicalization step of CFM-SE that converts `if-then` to `if-then-else` form which allows it to merge `if-then` branches. Also, instructions are shown on separate lines (Lines 4-7) for better readability. Figure 4.2b shows the code after dead code insertion. Here `else` path is empty therefore all the instructions are unaligned. `else` path contains the dead instructions inserted. For example, load operation in line 4 is repeated in line 10 after the dead code insertion. CFM-SE also tries to preserve def-use chains and minimize select operations needed for merging. For example, instruction `t7` at line 12 uses instruction `t6` at line 11. This is similar to instruction `t3` using `t2` as its first operand. `t7` uses `0` as its second operand to avoid any overflow/underflow bugs. This example also demonstrates how store instructions are handled during dead code insertion. On `if` path there is a store (Line 8) of value `t3` to `text[i]`. On `else` path the same store is performed (Line 14) but the stored value is `text[i]` (*i.e.* `t8`). This requires loading `text[i]` before the store (Line 13). This also inserts a redundant load to the `if` path (Line 7) to make the alignment complete. Figure 4.2c shows the code after the merging step. Extra select operations (shown as `C` ternary operator) are inserted to select operands if input operands do not match. In this example, it is safe to execute all the memory operations unconditionally and therefore the transformation is semantics-preserving. Transformed program is much faster to execute in KLEE compared to the original (Section 4.3).

4.4.2 Properties of CFM-SE Transformation:

Consider a single application of CFM-SE to a branch in program P. Assume program P^d is obtained after inserting dead instructions and P' is the final result of the transformation. Then the following properties hold:

1. Any dead instruction added to P^d is not used by original instructions in P^d.
2. Any dead store operation inserted into P^d does not mutate the memory space because the stored value is the same as the existing value in the memory.

3. Values of the dead instructions can not flow outside the merged branch because they will be filtered out by the ϕ-nodes at the end of the branch.

4. Transformation $P^d \rightarrow P'$ does not violate any program semantics because DARM [88] is a semantics-preserving transformation. Therefore, any failure that exists in P^d must exist in P'.

Property ①, ② ensures that dead code insertion does not alter the original computation of the program in any way. Therefore, if the original program has some bug, then program after inserting dead code will also have the same bug. These properties along with property ④ ensures the failure preservability of CFM-SE transformation. Property ③ ensures that if a bug is introduced in $P \rightarrow P^d$ transformation, then it will be realized within the merged region. This property allows us to undo the transformation and filter out false positive bugs introduced by CFM-SE. We describe the design of our false positive detection framework next.

4.4.3 False Positive Detection

As described in Section 4.4.2, assume the program we are testing using DSE is P. Let P' be the program after CFM-SE transformation. Let α_{crash} be a program input that causes a crash in P'. If executing the same input on P does not cause a crash, then we have a false positive bug. As discussed in Section 4.4.1, CFM-SE can introduce new bugs in the program, therefore detecting false positives is important. False positive bugs inserted by CFM-SE transformation must be realized within the region of the code where the transformation is applied. Dead instructions inserted by CFM-SE are not used by any of the original instructions. Therefore, their values can not flow outside the merged branch. In other words a false positive bug added by CFM-SE must materialize at an instruction produced by the transformation. An example would be a memory out of bounds access caused by unconditional execution of memory instructions. This bug will realize at the program location of this memory access. We can find this program location and avoid applying CFM-SE to that location and re-execute the program symbolically. This will avoid that specific false positive bug from occurring again.

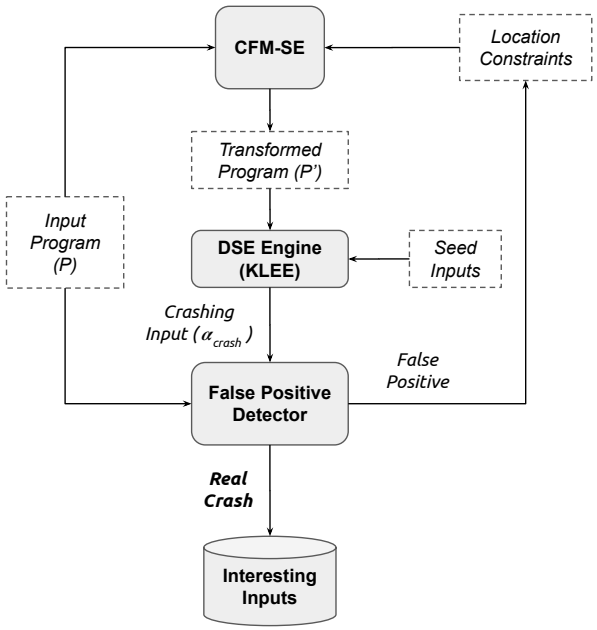

Figure 4.3. Symbolic execution driver loop used for detecting false positive bugs introduced by CFM-SE.

The false positive detection and re-execution driver is shown in Figure 4.3. Driver start by symbolically executing the CFM-SE transformed program P' in KLEE. If a crashing input (α_{crash}) is detected during symbolic execution, execution is stopped and the driver checks if α_{crash} is valid crash. This can be done by re-executing untransformed P with α_{crash}. If it is a real crash α_{crash} is collected as an interesting input. Otherwise, driver obtain the program location where the crash occurred and update location constraints for the CFM-SE transformation (*i.e.* CFM-SE it not applied to that location). Program P is recompiled with updated location information and driver loop continues. Restarting the whole DSE process can be expensive specifically if the false positive bug is found deep inside the program. Now the KLEE needs explore all previously explored paths again with recompiled program losing all the progress we made in the previous DSE exploration. To avoid the overhead of re-exploring the whole program from scratch, we use the inputs collected during the previous DSE run as seeds for the new DSE run.

4.4.4 Symbolic Variable Analysis

```
1  int foo(int x, int y) {
2    if (x > y) return x;
3    if (y > 0) return y;
4    return x + y;
5  }
6  int main() {
7    int a;
8    klee_make_symbolic(&a, sizeof(a), "a");
9    ...
10   int p = foo(a, 10);
11   ...
12   int q = foo(-5, a);
13   ...
14 }
```

Figure 4.4. Symbolic variable analysis example. Function `main` contains two calls to function `foo` with symbolic arguments. Depending on the call site of `foo` different instructions inside `foo` must be marked symbolic.

In DSE, the outcome of a conditional branch can depend on a symbolic variable. Such branches are called *symbolic branches*. If both outcomes of a symbolic branch (*true* and *false*) are feasible the DSE has to fork the execution and explore both outcomes of the branch. To minimize the number of paths explored by DSE, we need to apply the CFM-SE transformation only at symbolic branches that are expensive for DSE to explore. This requires identifying symbolic branches in the program at compile time. Our symbolic variable analysis is based on LLVM's divergence analysis [42]. Divergence analysis identifies variables that are data or control-dependent on divergent sources such as thread identifier in GPU programs. Divergence analysis is intra-procedural. It does not consider symbolic value flow across function boundaries resulting in significant loss of precision for our use case. Next we describe how we address this limitation to design an inter-procedural symbolic variable analysis.

First step of symbolic variable analysis is identifying symbolic sources. Symbolic sources simply refer to variables that are explicitly marked as symbolic by the user. Symbolic source can be a variable that is explicitly marked as symbolic by the user. For example, in

KLEE [100] a variable can be marked as symbolic using the `klee_make_symbolic` function. Any user arguments to the program are also considered symbolic sources (*i.e.* the arguments to the `main` function). Due to the complexity of reasoning about aliasing between different memory accesses from different functions, we also conservatively mark all memory loads of a function as symbolic as well. We observe that without this simplifying assumption, most functions deep down in the call graph are never marked to have any symbolic instructions. This approach is similar to LLVM divergence analysis [42] where any memory access from the global memory is assumed to be divergent.

Similar to divergence analysis, if we use an intra-procedural data flow analysis after marking symbolic sources, it will only mark a subset of true symbolic instructions in the program. This is because the symbolic property of a variable is not propagated to callees from a call site with symbolic arguments. Consider the example program in Figure 4.4. An intra-procedural symbolic variable analysis will identify the `a` variable in the `main` function as a symbolic source first and the mark call sites at lines 10 and 12 as symbolic due data dependance on `a`. However, none of the instructions in the `foo` function will be marked as symbolic because function foo does not have any explicit symbolic sources and symbolic variables are not propagated to callees at their call sites. We address this limitation by updating the symbolic sources of the callees at each call site and re-processing the callee if there is a change in the symbolic sources. The algorithm for inter-procedural symbolic variable analysis is shown in Algorithm 4.

First we mark the symbolic sources for all the functions in the program and insert each function into a work list. Then we process each function in the work list and propagate the symbolic property to other instructions within the function based on data or sync-dependances (similar to divergence analysis). Once the function is processed we check each call site in the function. If the call site is marked as symbolic, that means at least one argument of the function is symbolic. If that argument is not already marked symbolic we proceed to mark it as symbolic and insert that function into the work list for re-processing. This analysis is inter-procedural but context-insensitive. Context-sensitive analysis is not required because we are only interested in applying the CFM-SE transformation at symbolic branches at compile time. In the example program (Figure 4.4), processing main function

initially will mark call sites at lines 10 and 12 as symbolic. Then re-processing of function `foo` will mark all operations in the function as symbolic (lines 2, 4 because of input argument `x` and line 3 because of argument `y`). For function with variable number of arguments (*variadic functions*) [112] we mark all accesses to the variable arguments as symbolic if at least one of them is found to be symbolic at a call site.

Algorithm 4: Symbolic Variable Analysis Algorithm

```
Input: Program P
Output: Symbolic variable analysis result R
Set WL ← ∅
Map R ← ∅
for each function F in P do
    R ← updateSymbolicSources(F, R)
    WL.insert(F)
end
while WL ≠ ∅ do
    F ← WL.removeOneEelement()
    propagateSymbolic(F, R)
    for each call site C in F do
        if C is not marked as symbolic then
            continue
        end
        CF ← callee of C
        R ← updateSymbolicSources(CF, R)
        if CF is not in WL then
            WL.insert(CF)
        end
    end
end
return R
```

4.5 Evaluation

4.5.1 Implementation

We implemented the CFM-SE as an LLVM-IR[2] transformation pass. We extended the DARM [88] framework to implement the CFM-SE analysis and transformations described in section 4.4. Unlike DARM, our implementation only targets single basic block `if-then` con-

[2]↑LLVM-14.0.0

structs or diamond shaped `if-then-else` constructs. Since CFM-SE completely eliminates the branch, recursive application of CFM-SE can merge complex `if-then-else-if` chains as well. Merging transformation also keeps track of the original source line number information of the merged instructions. For example, if two instructions i_1 and i_2 with source line numbers l_1 and l_2 are merged, the merged instruction i_m will be attached additional debug information that contains the source line numbers l_1 and l_2. This helps us to keep compute the line coverage of the transformed program. We refer to this as *merged line coverage*. For the symbolic execution we used KLEE[3] DSE engine. CFM-SE pass is run as a pre-processing step before the symbolic execution begins in KLEE.

4.5.2 Experimental Setup

For all the experiments we used an X86 machine with 256 GB of memory and AMD Ryzen 64-Core Processor running Ubuntu 18.04.6 LTS. To evaluate the utility of CFM-SE transformation we designed out experiments to answer the following research questions:

- **DSE Performance (RQ1):** How effective is CFM-SE's branch elimination transformation in reducing the number of solver calls and mitigating the path explosion problem?
- **Bounded Verification (RQ2):** Can CFM-SE make bounded verification faster?
- **Coverage (RQ3):** Can a program transformed with CFM-SE achieve higher line coverage within a given time budget?

For RQ1 and RQ2, we use a benchmark suite of 11 programs consisting of well-known graph algorithms, sorting algorithms and our motivating example (*i.e.* `toupper`) from Section 4.3. The functionality of each benchmark and the nature of their inputs are described in Table 4.1. We selected these benchmarks because they are smaller enough so that KLEE can enumerate all feasible execution paths for sufficiently smaller input sizes within a reason-

[3]↑KLEE-2.3

able amount of time (*i.e.* within hours). This allows us to evaluate how effectively different techniques can mitigate the path explosion problem compared to vanilla KLEE.

Table 4.1. Description of the benchmarks used for RQ1 and RQ2

Benchmark	Description
toupper	converts all lowercase letters to uppercase in a fixed length `char` array
bitonic sort	use bitonic mergesort to sort an integer array [50]
connected components	computes the number of connected components in a graph represented in adjacency matrix form using Shiloach-Vishkin algorithm [113]
prim	finds a minimum spanning tree for weighted undirected graph
kruskal	finds a minimum spanning forest for a edge-weighted undirected graph
merge sort	recursive top-down merge sort for sorting an integer array
transitive closure	computes the reachability from vertex i to vertex j for all vertex pairs (i, j) in a directed graph
dilation	applies morphological dilation to a binary image over a 3×3 neighborhood [114]
detect edges	applies the Sobel-Feldman operator [115] to an input image to for edge detection
floyd warshall	finds the shortest paths between all pairs of vertices in a weighted directed graph [116]
erosion	applies morphological erosion to a binary image over a 3×3 neighborhood [114]

4.5.3 DSE Performance (RQ1)

For RQ1, we execute the 11 programs with symbolic inputs of different sizes. For the comparison we consider the following approaches:

- **Vanilla KLEE (K):** KLEE with default optimization settings.
- **KLEE with State Merging (SM) :** KLEE with state merging enabled using its built-in state merging mechanism [100]. For this, we manually instrumented the benchmarks to surround each symbolic `if-then` and `if-then-else` regions with `klee_open_merge` and `klee_close_merge` calls. This enables dynamic state merging of all states forked after the `klee_open_merge` call at a `klee_close_merge` call.

- **KLEE with CFM-SE (C):** KLEE with CFM-SE transformation enabled.
- **KLEE with CFM-SE and State Merging (C-SM):** KLEE with CFM-SE transformation and state merging enabled.

Because CFM-SE (**C**) is a compile time technique, its applicability is limited compared to dynamic state merging (**SM**). There for some programs, it is more beneficial to apply both techniques (**C-SM**). **C-SM** has the benefits of both CFM-SE and state merging *i.e.* it can eliminate branches at compile time and merge states at runtime. We use STP solver **stp** as the solver backend in KLEE. For each KLEE execution we use a time budget of 1 hour (`-max-time=3600s`) and memory budget of 50 GBs (`-max-memory=51200`). We also used the option `-only-output-states-covering-new` that makes KLEE only output the states that cover new code paths. For each run, we used `klee-stats` **klee_stats** tool to collect time KLEE takes to explore all possible program paths (or timeout), number of solver calls, average solver query size, and number of explored program paths. To reduce the noise in time measurements, we repeated each experiment 5 times and report the median.

Run Time : Table 4.2 shows the results of the experiments. For all the benchmarks, application of CFM-SE does not result in any crashes *i.e.* applying CFM-SE is *safe* for these benchmarks. Out of the 33 benchmark and input size configurations, approach **K** times out in 19 cases, **SM** times out in 4 cases. Both **C** and **C-SM** times out in only 3 cases. The 3 cases where **C** and **C-SM** time out occurs in kruskal and merge sort. In both these benchmarks, there are no opportunities for CFM-SE to eliminate branches *i.e.* both benchmarks contain either branches with symbolic array addresses. So the performance of **C** is similar to **K**. However, manual application of state merging (**SM**) still merges states in these two benchmarks yielding mixed results. For kruskal **SM** finishes the exploration faster than **K** for inputs sizes 3 and 4. However, for merge sort merging states with symbolic addresses makes things worse resulting in ≈ 5 times slow down for input size 5. In kruskal, even though diverging program paths contain memory accesses with symbolic addresses, the memory accesses are always to the same array element. Therefore, KLEE does not require additional solver calls for resolving the memory accesses.

Table 4.2. KLEE symbolic execution statistics collected for the approaches **K**, **C**, **SM** and, **C-SM**. Table shows the execution time, number of queries, average query size and, number of explored paths for the different benchmarks and inputs sizes. OOT = out of time (1 hour limit)

Benchmark	Input Size	Time(s)				Number of Queries				Average Query Size				Explored Paths			
		K	**C**	**SM**	**C-SM**	**K**	**C**	**SM**	**C-SM**	**K**	**C**	**SM**	**C-SM**	**K**	**C**	**SM**	**C-SM**
toupper	10	0.19	0.00	0.11	0.00	11	0	11	0	11	0	11	0	1,024	1	11	1
	50	OOT	0.00	0.59	0.00	26	0	51	0	11	0	11	0	10,974,670	1	51	1
	100	OOT	0.00	1.22	0.00	26	0	101	0	11	0	11	0	10,199,525	1	101	1
bitonic sort	4	0.55	0.00	0.11	0.00	37	0	7	0	103	0	145	0	28	1	7	1
	8	OOT	0.00	1.56	0.00	106,764	0	25	0	415	0	1,233	0	66,409	1	25	1
	16	OOT	0.00	28.58	0.00	94,373	0	81	0	403	0	9,336	0	119,136	1	81	1
connected components	3	0.09	0.05	0.14	0.07	10	12	44	12	4	151	145	151	512	3	29	3
	4	81.45	0.11	9.92	0.10	17	20	118	20	4	434	8,964	434	65,536	4	71	4
	5	OOT	0.28	OOT	0.23	26	30	122	30	4	955	1,510,179	955	16,348,259	5	96	5
prim	4	16.13	0.07	1.14	0.07	386	28	69	28	217	118	899	118	53,392	1	37	1
	5	OOT	0.16	6.06	0.16	7,513	45	121	45	320	228	2,597	228	4,238,759	1	69	1
	6	OOT	1.59	26.55	1.60	9,539	66	187	66	290	370	6,039	370	3,567,833	1	111	1
kruskal	3	15.83	12.33	10.48	12.74	668	667	444	444	360	360	356	356	144	144	120	120
	4	877.25	728.09	473.26	547.12	20,919	20,897	8,581	8,577	574	574	596	597	5,808	5,808	2,832	2,832
	5	OOT	OOT	OOT	OOT	49,050	53,409	2,390	2,436	648	654	625	630	18,903	20,364	797	809
merge sort	5	1.67	1.86	8.10	8.12	120	120	568	568	146	146	1,118	1,117	120	120	119	119
	10	OOT	OOT	OOT	OOT	113,712	103,186	92,071	90,878	385	382	2,647	2,636	115,481	106,063	24,284	23,571
	15	OOT	OOT	OOT	OOT	46,620	45,603	123,616	122,377	365	362	1,081	1,102	1,828,398	1,616,690	50,503	49,652
transitive closure	3	3.08	0.00	0.34	0.00	772	0	27	0	164	0	913	0	49	1	24	1
	4	394.77	0.00	1.43	0.00	75,154	0	64	0	309	0	4242	0	2,041	1	60	1
	5	OOT	0.00	4.44	0.00	141,506	0	125	0	389	0	13608	0	121,728	1	120	1
dilation	4	0.44	0.25	0.36	0.31	42	28	28	24	40	14	59	15	81	16	9	5
	5	6.24	0.55	0.77	0.54	240	52	57	43	130	15	129	15	11,502	512	19	10
	6	OOT	33.44	1.51	0.87	1,008	84	98	68	161	15	221	16	3,671,111	65,536	33	17
detect edges	3	OOT	0.01	5.18	0.00	26	2	12	2	323	8	561	8	23	2	19	2
	4	OOT	0.01	21.04	0.01	20	2	39	2	293	8	2,820	8	17	2	50	2
	5	OOT	0.01	47.82	0.01	20	2	84	2	293	8	6,657	8	17	2	99	2
floyd warshall	3	OOT	0.00	0.50	0.00	1,976	0	27	0	329	0	789	0	1,619	1	23	1
	4	OOT	0.00	2.75	0.00	79,222	0	64	0	507	0	3,915	0	60,274	1	58	1
	5	OOT	0.00	13.59	0.00	84,458	0	125	0	518	0	13,178	0	65,547	1	117	1
erosion	4	1.78	0.2	0.44	0.25	130	28	32	24	255	15	172	15	59	16	9	5
	5	148.57	0.46	1.04	0.43	7,347	52	61	43	598	15	401	16	3,516	512	19	10
	6	OOT	34.15	2.31	0.68	87,840	84	100	68	864	15	885	16	95,697	65,536	33	17

In fact the number of solver calls reduces for kruskal when **SM** is applied compared to **K** (444 vs 668 for input size 3). However, for merge sort, the symbolic memory accesses are to different array elements and KLEE requires additional solver calls to resolve the memory accesses. In this case solver calls increases significantly for **SM** compared to **K** (568 vs 120 for input size 5). For both these benchmarks, behavior of **C-SM** is similar to **SM** because CFM-SE does not apply to them. Overall, **C** is always faster than **K** for all the benchmarks and input sizes. Erosion and dilation are the only other benchmarks where **SM** outperforms **C** (for the largest input size). These two programs contains loops that are guarded by symbolic conditions. CFM-SE cannot eliminate branches containing control-flow regions (*i.e.* loops). However, state merging does not have this restriction and can still merge states

in these loops. Combining CFM-SE and state merging (**C-SM**) yields the best performance for these two benchmarks because it can eliminate branches to remove the solver overhead and merge states in locations where CFM-SE cannot eliminate branches.

Solver Calls : CFM-SE transformation is highly effective in reducing the number of solver calls for all the cases where it is applicable. Out of the 9 benchmarks that CFM-SE transformation is applicable, it reduces the number of solver calls in all of them. In toupper, bitonic sort, transitive closure, and floyd warshall, CFM-SE completely eliminates all symbolic branches resulting in zero solver calls during symbolic execution. Essentially, the CFM-SE-transformed program computes a single disjunctive path constraint representing all possible execution paths. Even though this is better for the performance, eliminating all symbolic branches can result in more expensive solver calls if this path constraint is used in a later symbolic branch. We investigate the impact of aggressive branch elimination in Section 4.5.4 and ways to mitigate its drawbacks in Section 4.6. Table 4.2 also shows the number of program paths explored by each technique. This metric is also quite similar to the number of solver calls. **C** end up exploring only a single program path in 5 of the benchmarks. Essentially, in these benchmarks **C** computes a single conjunctive path condition that represent any valid input to the program.

Solver Query Size : Average query size in Table 4.4 measures the average number of constructs per query issued to the solver. This is highly sensitive counter example caching mechanism used in KLEE [100]. If the constraints encountered during different program paths are similar number cache hits increase and therefore number of newly created queries goes down. If cache hits are low, then average query size increases because now more different queries are going to the solver. According to our evaluation state merging negatively impact the caching mechanism in KLEE. Average query size is much higher for **SM** for most of the benchmarks. This is because merging states from different loop iterations create complex irregular constraints that are not repeated. In fact average query size is significantly higher for **SM** in 8 out of the 11 benchmarks considered. On the other hand, static merging of program paths reduces the average query size. This is because CFM-SE reduces the number

of solver calls significantly and, also helps to maintain the repeatability of constraints. We observe that **C** achieves more cache hits per program path explored compared to **SM** on several benchmarks.

Table 4.3. Query cache hits per explored path for benchmarks where **C** explores more than a single path.

Benchmark	**Input Size**	**Query Cache Hits / Explored Paths**		
		C	**SM**	**C-SM**
connected components	5	10.8	1.75	10.8
prim	6	110	1.98	110
merge sort	15	2.67×10^{-2}	3.70	3.73
dilation	6	1.95×10^{-3}	1.21×10^{-1}	2.12
detect edges	5	1	1.17	1
erosion	6	1.95×10^{-3}	3.03×10^{-2}	2.12

Regularity in Generated Constraints : To further investigate the impact of CFM-SE on the regularity of generated constraints, we measure the number of cache hits per program path explored for each technique. Table 4.3 shows the results of this experiment. We only considered benchmarks where after applying CFM-SE, DSE still requires solver calls during the exploration (*i.e.* number of paths explored is greater than 1). If the path constraints are repeated on different program paths, then we can expect more cache hits per program path. Because dynamic state merging can merge any two random states, it can generate more complex irregular constraints as the path condition. This results in reduced cache hits per program path. This is evident in 4 out of the 6 benchmarks considered (*i.e.* connected components, prim, dilation and erosion). In detect edges applying CFM-SE merges all paths inside program loops and therefore, DSE explores a fixed number of paths regardless of the input size. In that case, **SM** has slightly higher cache hits per program path compared to **C** (or **C-SM**). For merge sort cache hits per program path increases with state merging because of the drastic increase in array queries and most of these queries hitting the cache. This study shows that CFM-SE can merge program states statically while maintaining the

regularity of generated constraints enabling better query caching. In contrast, dynamic state merging can merge any two random states and therefore, it can generate complex irregular constraints that are may not be repeated.

4.5.4 Bounded Verification (RQ2)

Table 4.4. Time spent and number of solver calls issued by KLEE for benchmarks instrumented with verification conditions. Table shows the statistics for different techniques and input sizes considered. OOT = out of time (1 hour time limit)

Benchmark	Input Size	Time(s)				Queries			
		K	**C**	**SM**	**C-SM**	**K**	**C**	**SM**	**C-SM**
toupper	10	0.45	0.13	0.24	0.13	21	11	21	11
	50	OOT	0.63	1.24	0.66	26	51	101	51
	100	OOT	1.27	2.55	1.28	26	101	201	101
bitonic sort	4	1.37	0.45	0.55	0.45	121	4	10	4
	8	OOT	104.94	104.68	100.88	150,247	8	32	8
	16	OOT	OOT	OOT	OOT	98,285	3	83	3
dilation	4	0.59	0.31	0.44	0.42	58	48	44	44
	5	27.06	1.22	0.97	0.77	270	92	82	77
	6	OOT	264.85	1.87	1.3	1,322	144	134	120
erosion	4	2.35	0.29	0.55	0.38	181	64	48	44
	5	245.74	1.48	1.4	0.72	13,455	142	86	77
	6	OOT	286.65	3.27	1.19	94,175	228	136	120

In RQ1, we only focused on the functional correctness of the benchmarks (*i.e.* benchmark does not crash all possible inputs) and how effective each technique is on mitigating the path explosion problem. In some benchmarks, aggressive branch elimination merge all possible execution paths into a single program path and corresponding path condition is not used in any of the program branch. This can be observed in 4 benchmarks (*i.e.* toupper, bitonic sort, transitive closure, and floyd warshall) where **C** has 0 solver calls and end up exploring only a single program path. In RQ2, we inserted assertions at the end of the program to verify if the output satisfy certain conditions that are known to be true after the program execution.

Constructing correctness assertions for all the benchmark programs is non-trivial and checking the assertions is computationally expensive in some cases (*i.e.* graph algorithms). Therefore, for RQ2 we consider a subset of the benchmarks with the listed assertions.

- *toupper:* check if the output string contains only upper case characters.
- *bitonic sort:* check if the output array is sorted in ascending order.
- *erosion:* check if each pixel value in the output binary image is less than or equal to the corresponding pixel value in the input image.
- *dilation:* check if each pixel value in the output binary image is greater than or equal to the corresponding pixel value in the input image.

We modified the programs to check the above conditions using `klee_assert` statements and measured the time and number of queries taken for the different approached considered in RQ1. Table 4.4 shows the results of the experiments. In general results suggest the **C** is very effective in reducing the number of solver calls and improving the DSE performance. In toupper, **C** and **C-SM** are the best performing approaches having close to 2× reduction in both runtime and number of solver calls compared to **SM**. In bitonic sort, the performance of **C** and **SM** have comparable performance even though **SM** has significantly more solver calls (8 vs 32). This benchmark contains memory reads/write with loop carried dependances (*i.e.* current iteration uses the values written to memory by the previous iteration). Branch elimination on such loops makes the path constraints more complex because the conditional assignments are converted complex `ite` containing different values in memory. Even though the number of solver calls are reduced by **C**, the solver calls are more expensive because of the complex path constraints. **SM** also merge the constraints but state merging is sensitive to the path exploration strategy in use *i.e.* it does not necessarily merge states forked within the same iteration of a loop. Because of this reason **SM** end up exploring more paths with less complex constraints compared to **C**. For input size 16 in bitonic sort all the approaches time out. For dilation and erosion, the best approach is **C-SM** because these two benchmarks has opportunities for both branch elimination and state merging (Section 4.5.3). **SM** works better than **C** for both the benchmarks because of its ability to merge states from arbitrary

control-flow paths whereas **C** is limited to merging condition branches containing straight-line code. As evident by the results, **C-SM** is significantly better than **SM** in terms of runtime and number of solver calls. This shows the utility of transformations like CFM-SE in improving the performance of DSE.

4.5.5 Coverage (RQ3)

Branch elimination allows KLEE to avoid the solver overhead and reach unexplored program locations faster. Essentially, this helps KLEE to achieve more coverage in less time. To assess how well CFM-SE transformation helps KLEE in achieving this goal, we consider 3 real-world subjects: `GNU oSIP-4.0.0` (*i.e.* `libosip`) [117], `GNU libtasn1-2.11` [118], and `chcon` utility from `coreutils-6.11` [119]. `libosip` is a library for Voice Over IP (VoIP) applications. It implements the Session Initiation Protocol (SIP) and provides a programming interface for building SIP applications. `libtasn1` is a library for encoding data objects in a machine-neutral fashion according to the Abstract Syntax Notation (ASN.1) specification. `chcon` utility in `coreutils` is used to change the SELinux security of a file [120]. We use these benchmarks because they contain large complex codebases that can be compiled into LLVM-IR and, they have been used in similar KLEE-based studies related to DSE [100], [121]. For `libosip` and `libtasn1`, we use the benchmark setup used in Chopper [121]. This includes manually written test driver programs that initialize the library interfaces and invoke the library functions with symbolic inputs. In `libosip`, test driver mainly focuses on testing the `osip_message_parse` function with symbolic message of size 10 bytes. In `libtasn1`, driver tests the `asn1_der_decoding` function with a fixed sized (*i.e.* 32 bytes) symbolic input. For `chcon`, we followed the setup described in KLEE `coreutils` experiment [122].

For each benchmark considered, we run both KLEE and KLEE with CFM-SE using the driver program described in Section 4.4.3. We use the default KLEE options described in Chopper and `coreutils` experiments. For `libosip` and `libtasn1`, we use 3 hour time limits and for `chcon` we use 1 hour time limit. Each benchmark is compiled with `-g` flag to generate debug information (*i.e.* source line information) for the LLVM-IR. Figure 4.5 shows the source line coverage plotted against time for `libosip` benchmark. Line coverage is the

percentage of distinct source lines that have been explored so far out of the total distinct source lines covered by all the LLVM-IR instructions in the compiled program. In this benchmark the maximum line coverage that can be achieved is around 36%. This is because the test driver only focus on testing certain interfaces of the library and some functions are not invoked at all. Interestingly, CFM-SE can reach the maximum line within in less than 500 seconds while KLEE takes close to 1 hours to reach the same coverage. In KLEE, coverage improves slowly within approximately 1000 seconds to 3000 seconds.

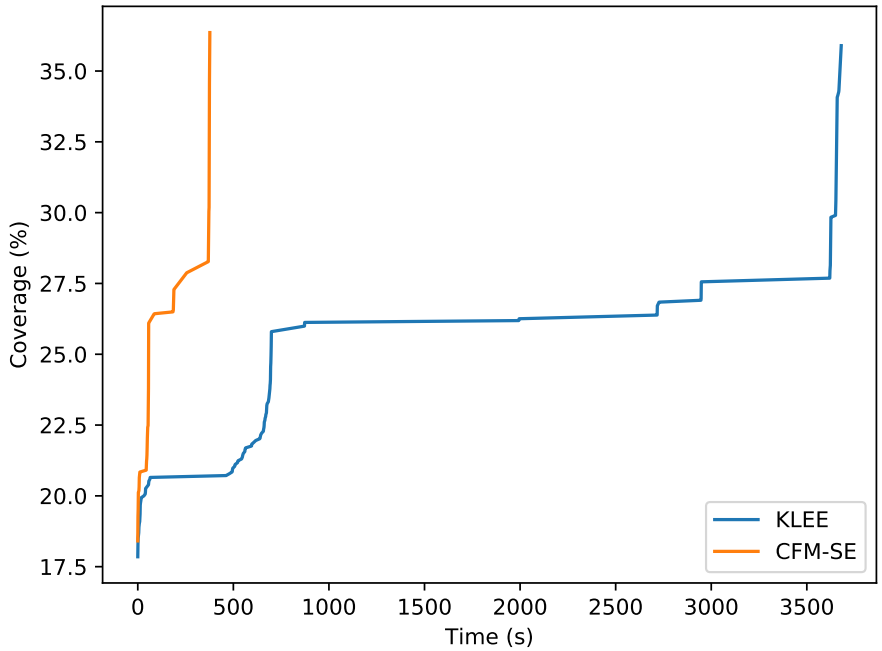

Figure 4.5. Source line coverage vs time for `libosip` benchmark

This is caused by symbolic execution getting stuck inside loops that have conditional branches or function calls with conditional branches. These loops do not have early exits so KLEE can not explore other paths outside the loop without finishing the whole loop execution. We found several loops in `libosip` that have this property. Aggressively branch elimination of branches inside these loops allows KLEE to finish the execution of the loops faster and explore more paths outside them. In this benchmark we did not find any false positive bugs that are introduced by CFM-SE. In other words, for the lines covered in

the benchmark CFM-SE transformation is safe. The fact that CFM-SE can achieve more coverage faster also means that the path queries generated by CFM-SE is not significantly more expensive than the path queries generated by KLEE.

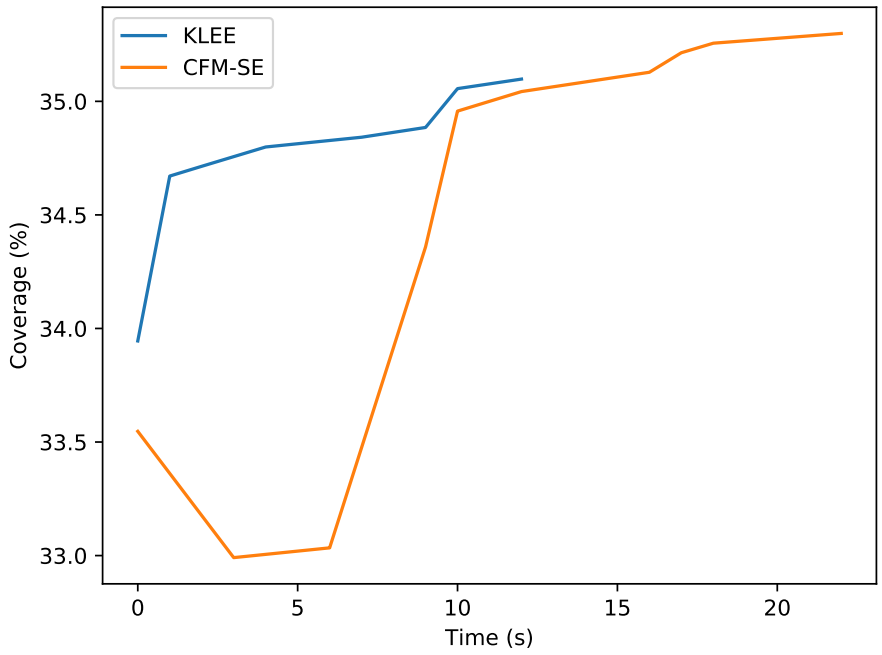

Figure 4.6. Source line coverage vs time for `libtasn1` benchmark

Figure 4.6 shows the source line coverage for `libtasn1` benchmark. In this benchmark, both KLEE and CFM-SE crash due to a `malloc` call with symbolic size. KLEE reach this location faster than CFM-SE (14 seconds vs 25 seconds). CFM-SE encounters 3 false positive bugs before reaching the `malloc` call. When a false positive is encountered, driver program relaunch KLEE with additional location constraints on CFM-SE transformation so that the transformation does not apply to the false positive location again. In this case all the false positive bugs are caused by out of bound memory accesses. These are caused by CFM-SE eliminating the branches that are guarding against out of bound memory accesses. The reduction in coverage for CFM-SE in Figure 4.6 is caused by the fresh re-execution of the program.

`chcon` benchmark also exhibits similar behavior as `libosip` benchmark (Figure 4.7). CFM-SE does not introduce any spurious bugs in this benchmark and achieves more coverage faster than KLEE. Maximum source line coverage achieved in this benchmark is approximately 64%. DARM achieves this coverage in around 2200 seconds while KLEE takes around 3400 seconds to reach the same coverage.

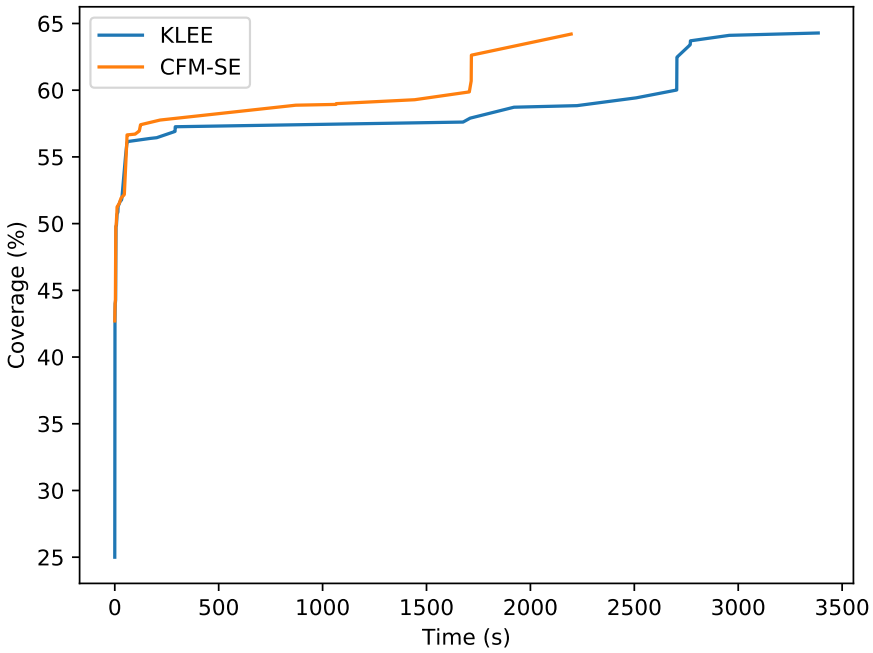

Figure 4.7. Source line coverage vs time for `chcon` benchmark

4.6 Limitations of CFM-SE

4.6.1 Constraint Complexity

There are several limitations that restrict the applicability and generality of CFM-SE. Unlike DARM (Chapter 2) and CFM-CS (Chapter 3), CFM-SE does not use a cost model to reason about the profitability of the branch elimination. CFM-SE uses simple heuristics based on the symbolic variable analysis to decide whether to apply the branch elimination or not. For example, if the branch contains memory accesses with symbolic addresses, CFM-

SE does not apply the branch elimination. CFM-SE does not reason about how complex the constraints can become if they are used in a future solver query. This requires estimating the complexity of queries that could be generated by a given sequence of instructions. Statically estimating the query cost has been explored by previous work related to state merging [26]. However, estimating the query cost of compile-time transformation is not well-explored. We believe this is an interesting problem that can be explored in future work. The benefits of CFM-SE is sensitive to the structure of dependances in program loops. If all different paths that are possible in a loop does not matter for branches outside the loop, statically merging the branches inside the loop is highly beneficial. This kind of program behavior can be expected when the program loops *does not* contain loop carried dependances. We observe superior performance for benchmarks such as `toupper`, `dilation` and, `erosion` because of this reason (Section 4.5.4). In these benchmarks, constraints generated by CFM-SE*does not* grow on each iteration of the loop, and therefore the solver queries are not expensive. However, if the program loops contain loop carried dependances, the constraints generated by CFM-SE can grow on each iteration of the loop. This is evident in `bitonicsort` benchmark (Section 4.5.4). In this benchmark, CFM-SE's performance is comparable to state merging even though CFM-SE reduces the number of solver calls significantly. In other words, due to the loop carried dependances, the constraints generated by CFM-SE keeps getting more complex when the input size increases. This limits the scalability of CFM-SE for certain programs. However, DSE is not intended to be used for large input sizes where none of the techniques are scalable. This observation gives us good insights on designing a cost model for static evaluation of query cost.

4.6.2 Test Generation

Program transformed using CFM-SE has less program paths compared its untransformed version. With CFM-SE DSE will explore less program paths and generate less test cases. Also, CFM-SE might not generate test cases that cover specific program paths in the original program, due to static path merging. Therefore, if the goal is to achieve maximum possible coverage for a given program, CFM-SE might not be the best choice because it can not

generate test cases that cover all the paths in the original program. This limitation is common for any approach that tries to merge program states including state merging [26]. However, achieving maximizing possible coverage for larger programs is not a realistic goal due to path explosion. Static path merging capability of CFM-SE can make DSE reach new program locations faster and achieve more coverage within a limited time budget. This is a useful property for testing real-world programs.

4.6.3 General Applicability

We have developed CFM-SE to be used with KLEE-based dynamic symbolic execution. KLEE is designed to detect specific types of program bugs such as assertion violations, out of bound memory access, and memory leaks, out of bound vector accesses, executing unexpected instructions (*e.g.* `abort`), writing to read-only memory, division by zero etc. To trigger these bugs the test generation tool needs to find an input that reaches the program location where the bug is present. KLEE is an efficient tool to solve the reachability problem and generate an input if the program location is reachable. Given that CFM-SE transformation does not change the program semantics (*i.e.* all false positive bugs are filtered out), it can improve KLEE's ability to reach these bug triggering program locations faster. This is because CFM-SE is failure preserving, and it reduces the number of program paths that KLEE needs to explore to reach the bug triggering program location. However, we do not provide any guarantees that CFM-SE will help in detecting other types of bugs that are not related to the reachability problem. For example, CFM-SE might not be useful for detecting bugs related to concurrency and synchronization of parallel programs. Also, CFM-SE might not be useful for other test generation tools like fuzzing [123]. In coverage based greybox fuzzing, preserving program branches is important to provide the necessary feedback about the coverage into the fuzzing loop. For example, if a program crash requires a specific branch to be taken at an earlier program location, the fuzzing tool needs to generate an input that takes that branch. If the branch is eliminated by CFM-SE, the fuzzing tool might not be able to generate an input that takes that branch because fuzzer can not provide the necessary feedback about branch decision.

4.7 Related Work

4.7.1 Dynamic Techniques

Many attempts have been made to mitigate the path explosion problem by guiding DSE only on interesting program paths [19]. Function and loop summarization produces summaries of frequently executed code sections and reuse that to avoid path explosion [27], [28]. Path equivalence and subsumption based techniques works by avoiding redundant program paths that do not reveal new information [104], [105]. Under-constrained symbolic execution applies symbolic execution to functions or code regions by isolating them from the surrounding application[29]. Any constraints that are applied to tested function by external (*i.e.* global) sources are considered *under-constrained*. State merging [26] attempts to combine different program paths explored during symbolic execution together to avoid path explosion.

4.7.2 Compiler Techniques

Instead of improving the heuristics for guiding symbolic execution, application of targeted program transformations to improve the performance of symbolic execution is also a well-studied in the literature. Testability transformations is a type of program transformation that improves the ability of a given test generation method to generate tests for the original untransformed program. Prior work have shown that such transformations can improve the performance for test generation techniques [32] Collingbourne *et al.* used branch predication to convert symbolic branches into *ite* expressions, thereby reducing the number of explored program path exponentially [33]. Wagner *et al.* proposed *-OVERIFY*, a new compiler optimization switch (*i.e.* a collection of optimizations) that enables fast verification of programs [31]. Wagner used DSE as a case study to show that selective application of compiler optimizations like constant folding, loop unswitching, if-conversion can drastically reduce the time spent in verification. Cadar *et al.* argued that compiler optimizations must be first-class ingredient in a practical DSE platform [30]. Perry *et al.* proposed a semantics preserving program transformation to accelerate DSE on programs with array accesses [107].

Inserting dead code to improve test generation techniques have also been explored in compiler testing [124].

4.8 Conclusion

Dynamic Symbolic Execution (DSE) is a dynamic program analysis technique that executes a program with symbolic inputs. DSE can enumerate all feasible execution paths in a program and generate tests for each path explored. However, DSE suffers from path explosion problem where the number of program paths can grow exponentially with the number of branch points in the program. Many dynamic and compiler-based techniques have been proposed to mitigate the impacts of path explosion in DSE. In this work, we propose a novel non-semantics preserving compiler transformation called CFM-SE to improve the scalability of DSE. CFM-SE inserts additional code to divergent code paths at symbolic branches to make the computation sequences within the branch identical and, merge the divergent paths together to remove the symbolic branch. CFM-SE is not semantics-preserving therefore, it can introduce new bugs to the program. We provide a framework for detecting such false positive bugs that could be introduced by transformations like CFM-SE. Our evaluation of CFM-SE shows that it can reduce the number of solver queries significantly and, improve the runtime of DSE on several important benchmarks. We also show that CFM-SE can help achieve more code coverage faster in real-world applications.

5. CONCLUSION

In programs with irregular control-flow structures (such as conditional branches), the runtime behavior of the program depends on various factors like the input data and the execution environment (*i.e.* hardware). Optimizing such programs for performance, efficiency, and testability is a challenging task because of several reasons. First, predicting the runtime behavior of the program is difficult because of the complex interdependencies between the input data, program execution paths, and the execution environment. Second, the complex control-flow structures resulted by the presence of deeply nested branches can make the program less performant and hard to optimize for efficiency metrics like code size. Third, the complex control-flow structures can make the program hard to test using dynamic test generation techniques due to the large number of possible execution paths. To optimize programs with irregular control-flow, we need to find some form of regularity in its structure. In this dissertation, we explore how *code similarity* can be exploited to make programs with irregular control-flow more performant, efficient, and testable. First, we propose *Control-flow Melding* (*i.e.* DARM), a compiler transformation that can merge structurally similar control-flow regions containing similar computations in GPU programs to reduce control-flow divergence. Control-flow melding is capable of exploiting both structural similarity and instruction similarity to optimize programs and, it is more general than existing methods. We show that DARM is effective in reducing control-divergence and improving the performance of several important GPU applications on ADM GPUs. Next, we extend control-flow melding for code size reduction in CPU programs. We develop a new technique called CFM-CS, that detects control-flow regions with similar instruction sequences and, merges them to reduce code size. We evaluate CFM-CS in several well-known CPU benchmark suites and show that it can reduce code size without significant performance or compile-time overhead. Finally, we propose CFM-SE, a non-semantics preserving compiler transformation targeted at improving the scalability of dynamic symbolic execution (DSE). The key idea of CFM-SE is that static merging of program paths can greatly improve the scalability of DSE. CFM-SE statically determines which branches of a program can be expensive during DSE and, merge the divergent paths of the branch to remove the expensive branch. We also provide a false

positive detection framework to detect any spurious bugs that might be introduced by CFM-SE due to its non-semantics preserving nature. We evaluate CFM-SE on several important benchmarks and show that can significantly reduce the number of solver queries in DSE, improve the runtime of DSE, and increase the code coverage on large programs within a limited time budget.

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2012, pp. 141–151.

[2] H. Vo, "Hardware support for irregular control flow in vector processor," *May*, vol. 7, p. 10, 2012.

[3] S. Moll and S. Hack, "Partial control-flow linearization," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 543–556, ISBN: 9781450356985. DOI: 10.1145/3192366.3192413. [Online]. Available: https://doi.org/10.1145/3192366.3192413.

[4] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate gpu performance model for effective control flow divergence optimization," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IEEE, 2012, pp. 83–94.

[5] M. Rhu and M. Erez, "The dual-path execution model for efficient gpu control flow," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 591–602. DOI: 10.1109/HPCA.2013.6522352.

[6] W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient simt control flow," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 25–36. DOI: 10.1109/HPCA.2011.5749714.

[7] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A scalable multi-path microarchitecture for efficient gpu control flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 248–259. DOI: 10.1109/HPCA.2014.6835936.

[8] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15, Portland, Oregon: Association for Computing Machinery, 2015, pp. 489–501, ISBN: 9781450334020. DOI: 10.1145/2749469.2750410. [Online]. Available: https://doi.org/10.1145/2749469.2750410.

[9] J. Anantpur and G. R., "Taming control divergence in gpus through control flow linearization," in *Compiler Construction*, A. Cohen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 133–153, ISBN: 978-3-642-54807-9.

[10] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4, Newport Beach, California, USA: Association for Computing Machinery, 2011, ISBN: 9781450305693. DOI: 10.1145/1964179.1964184. [Online]. Available: https://doi.org/10.1145/1964179.1964184.

[11] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr., "Divergence analysis and optimizations," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 320–329. DOI: 10.1109/PACT.2011.63.

[12] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Efficient warp execution in presence of divergence with collaborative context collection," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48, Waikiki, Hawaii: Association for Computing Machinery, 2015, pp. 204–215, ISBN: 9781450340342. DOI: 10.1145/2830772.2830796. [Online]. Available: https://doi.org/10.1145/2830772.2830796.

[13] W.-K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, San Diego, CA, USA: Springer-Verlag, 2003, pp. 401–417, ISBN: 3540403256.

[14] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[15] Free Software Foundation, *GCC, the GNU compiler collection*, http://gcc.gnu.org, 2018.

[16] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, Orlando, Florida, USA: ACM, 1994, pp. 147–158. DOI: 10.1145/773473.178256.

[17] J. Cocke, "Global common subexpression elimination," in *Proceedings of a Symposium on Compiler Optimization*, New York, NY, USA: ACM, 1970, pp. 20–24. DOI: 10.1145/800028.808480.

[18] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen, "Graph-based procedural abstraction," in *International Symposium on Code Generation and Optimization (CGO'07)*, Mar. 2007, pp. 259–270. DOI: 10.1109/CGO.2007.14.

[19] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, May 2018, ISSN: 0360-0300. DOI: 10.1145/3182657. [Online]. Available: https://doi.org/10.1145/3182657.

[20] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, Lisbon, Portugal: Association for Computing Machinery, 2005, pp. 263–272, ISBN: 1595930140. DOI: 10.1145/1081706.1081750. [Online]. Available: https://doi.org/10.1145/1081706.1081750.

[21] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, Springer, 2008, pp. 351–366.

[22] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*, vol. 10, 2001, pp. 176–194.

[23] A. Valmari, "The state explosion problem," *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pp. 429–528, 2005.

[24] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *ACM SigPlan Notices*, vol. 48, no. 10, pp. 19–32, 2013.

[25] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings 18*, Springer, 2011, pp. 95–111.

[26] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: Association for Computing Machinery, 2012, pp. 193–204, ISBN: 9781450312059. DOI: 10.1145/2254064.2254088. [Online]. Available: https://doi.org/10.1145/2254064.2254088.

[27] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," ser. FSE 2016, Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 61–72, ISBN: 9781450342186. DOI: 10.1145/2950290.2950340. [Online]. Available: https://doi.org/10.1145/2950290.2950340.

[28] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08, Budapest, Hungary: Springer-Verlag, 2008, pp. 367–381, ISBN: 3540787992.

[29] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, Washington, D.C.: USENIX Association, 2015, pp. 49–64, ISBN: 9781931971232.

[30] C. Cadar, "Targeted program transformations for symbolic execution," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: Association for Computing Machinery, 2015, pp. 906–909, ISBN: 9781450336758. DOI: 10.1145/2786805.2803205. [Online]. Available: https://doi.org/10.1145/2786805.2803205.

[31] J. Wagner, V. Kuznetsov, and G. Candea, "-OVERIFY: Optimizing programs for fast Verification," in *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, Santa Ana Pueblo, NM: USENIX Association, May 2013. [Online]. Available: https://www.usenix.org/conference/hotos13/session/wagner.

[32] M. Harman, L. Hu, R. Hierons, *et al.*, "Testability transformation," vol. 30, no. 1, pp. 3–16, Jan. 2004, ISSN: 0098-5589. DOI: 10.1109/TSE.2004.1265732. [Online]. Available: https://doi.org/10.1109/TSE.2004.1265732.

[33] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11, Salzburg, Austria: Association for Computing Machinery, 2011, pp. 315–328, ISBN: 9781450306348. DOI: 10.1145/1966445.1966475. [Online]. Available: https://doi.org/10.1145/1966445.1966475.

[34] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 407–420. DOI: 10.1109/MICRO.2007.30.

[35] *HIP Programming Guide v4.1*, [Accessed 17-Dec-2021]. [Online]. Available: https://rocmdocs.amd.com/en/latest/.

[36] *CUDA C++ Programming Guide*, [Accessed 17-Dec-2021]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[37] M. Pharr and W. R. Mark, "Ispc: A spmd compiler for high-performance cpu programming," in *2012 Innovative Parallel Computing (InPar)*, 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[38] *Using cuda warp-level primitives*, [Accessed 17-Dec-2021]. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[39] *NVCC :: CUDA Toolkit Documentation*, [Accessed 17-Dec-2021]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[40] *ROCm Compiler SDK*, [Accessed 17-Dec-2021]. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Compiler-SDK.html.

[41] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991, ISSN: 0164-0925. DOI: 10.1145/115372.115320. [Online]. Available: https://doi.org/10.1145/115372.115320.

[42] R. Karrenberg and S. Hack, "Improving performance of opencl on cpus," in *Compiler Construction*, M. O'Boyle, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–20, ISBN: 978-3-642-28652-0.

[43] T. Schaub, S. Moll, R. Karrenberg, and S. Hack, "The impact of the simd width on control-flow and memory divergence," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, Jan. 2015, ISSN: 1544-3566. DOI: 10.1145/2687355. [Online]. Available: https://doi.org/10.1145/2687355.

[44] R. Karrenberg and S. Hack, "Whole Function Vectorization," in *International Symposium on Code Generation and Optimization*, ser. CGO, 2011. DOI: 10.1109/CGO.2011.5764682. [Online]. Available: http://www.cdl.uni-saarland.de/papers/karrenberg_wfv.pdf.

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

[46] T. Lloyd, K. Ali, and J. N. Amaral, "Gpucheck: Detecting cuda thread divergence with static analysis," Deparment of Computer Science, University of Alberta, Tech. Rep., 2019. DOI: https://doi.org/10.7939/R3W669R4S. [Online]. Available: https://era.library.ualberta.ca/items/7ab2b28d-b111-448f-8273-2ff219132908.

[47] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Formalizing structured control flow graphs," in *Languages and Compilers for Parallel Computing*, C. Ding, J. Criswell, and P. Wu, Eds., Cham: Springer International Publishing, 2017, pp. 153–168, ISBN: 978-3-319-52709-3.

[48] D. Sampaio, R. M. d. Souza, C. Collange, and F. M. Q. Pereira, "Divergence analysis," vol. 35, no. 4, Jan. 2014, ISSN: 0164-0925. DOI: 10.1145/2523815. [Online]. Available: https://doi.org/10.1145/2523815.

[49] J. Rosemann, S. Moll, and S. Hack, "An abstract interpretation for spmd divergence on reducible control flow graphs," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. DOI: 10.1145/3434312. [Online]. Available: https://doi.org/10.1145/3434312.

[50] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference (AFIPS '68 (Spring))*, 1968, pp. 307–314. DOI: 10.1145/1468075.1468121.

[51] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *ACM J. Exp. Algorithmics*, vol. 14, Jan. 2010, ISSN: 1084-6654. DOI: 10.1145/1498698.1564500. [Online]. Available: https://doi.org/10.1145/1498698.1564500.

[52] *llvm::RegionBase Class Template Reference*, [Accessed 17-Dec-2021]. [Online]. Available: https://llvm.org/doxygen/classllvm_1_1RegionBase.html.

[53] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," *SIGPLAN Not.*, vol. 29, no. 6, pp. 171–185, Jun. 1994, ISSN: 0362-1340. DOI: 10.1145/773473.178258. [Online]. Available: https://doi.org/10.1145/773473.178258.

[54] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981, ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(81)90087-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283681900875.

[55] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Function merging by sequence alignment," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019, pp. 149–163. DOI: 10.1109/CGO.2019.8661174.

[56] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, and H. Leather, "Effective function merging in the ssa form," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 854–868, ISBN: 9781450376136. DOI: 10.1145/3385412.3386030. [Online]. Available: https://doi.org/10.1145/3385412.3386030.

[57] *llvm-mca - LLVM Machine Code Analyzer*, [Accessed 07-March-2022]. [Online]. Available: https://www.llvm.org/docs/CommandGuide/llvm-mca.html.

[58] *CostModel.cpp File Reference*, [Accessed 17-Dec-2021]. [Online]. Available: https://llvm.org/doxygen/CostModel_8cpp.html.

[59] *llc - LLVM static compiler*, [Accessed 17-Dec-2021]. [Online]. Available: https://llvm.org/docs/CommandGuide/llc.html.

[60] E. Herruzo, G. Ruiz, J. I. Benavides, and O. Plata, "A new parallel sorting algorithm based on odd-even mergesort," in *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*, 2007, pp. 18–22. DOI: 10.1109/PDP.2007.10.

[61] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[62] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[63] *CUDA Samples*, [Accessed 17-Dec-2021]. [Online]. Available: https://docs.nvidia.com/cuda/cuda-samples/.

[64] A. Kerr, G. Diamos, and S. Yalamanchili, "A characterization and analysis of ptx kernels," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 3–12. DOI: 10.1109/IISWC.2009.5306801.

[65] *ROCm-Developer-Tools / rocprofiler*, [Accessed 17-Dec-2021]. [Online]. Available: https://github.com/ROCm-Developer-Tools/rocprofiler.

[66] *NVIDIA TESLA V100 GPU ARCHITECTURE*, [Accessed 28-Feb-2022]. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[67] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970, ISSN: 0022-2836. DOI: https://doi.org/10.1016/0022-2836(70)90057-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022283670900574.

[68] H. Wu, G. Diamos, J. Wang, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in bulk synchronous gpu applications," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 170–185, May 2012, ISSN: 1094-3420. DOI: 10.1177/1094342011434814. [Online]. Available: https://doi.org/10.1177/1094342011434814.

[69] J. Fukuhara and M. Takimoto, "Branch divergence reduction based on code motion," *Journal of Information Processing*, vol. 28, pp. 302–309, 2020. DOI: 10.2197/ipsjjip.28.302.

[70] O. Rüthing, J. Knoop, and B. Steffen, "Sparse code motion," ser. POPL '00, Boston, MA, USA: Association for Computing Machinery, 2000, pp. 170–183, ISBN: 1581131259. DOI: 10.1145/325694.325715. [Online]. Available: https://doi.org/10.1145/325694.325715.

[71] S. Damani, D. R. Johnson, M. Stephenson, *et al.*, "Speculative reconvergence for improved simt efficiency," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020, San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 121–132, ISBN: 9781450370479. DOI: 10.1145/3368826.3377911. [Online]. Available: https://doi.org/10.1145/3368826.3377911.

[72] S. Damani and V. Sarkar, "Common subexpression convergence: Aănew code optimization for simt processors," in *Languages and Compilers for Parallel Computing*, S. Pande and V. Sarkar, Eds., Cham: Springer International Publishing, 2021, pp. 64–73, ISBN: 978-3-030-72789-5.

[73] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 235–246, Jun. 2010, ISSN: 0163-5964. DOI: 10.1145/1816038.1815992. [Online]. Available: https://doi.org/10.1145/1816038.1815992.

[74] *MergeFunctions pass, how it works*, [Accessed 4-Mar-2022]. [Online]. Available: https://www.llvm.org/docs/MergeFunctions.html.

[75] D. Rayside, E. Mamas, and E. Hons, "Compact java binaries for embedded systems," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99, Mississauga, Ontario, Canada: IBM Press, 1999, p. 9.

[76] S. Liao, S. Devadas, and K. Keutzer, "A text-compression-based method for code size minimization in embedded systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 1, pp. 12–38, Jan. 1999, ISSN: 1084-4309. DOI: 10.1145/298865.298867. [Online]. Available: https://doi.org/10.1145/298865.298867.

[77] G. Liu, U. Farooq, C. Zhao, X. Liu, and N. Sun, "Linker code size optimization for native mobile applications," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023, Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 168–179, ISBN: 9798400700880. DOI: 10.1145/3578360.3580256. [Online]. Available: https://doi.org/10.1145/3578360.3580256.

[78] R. Lavaee, J. Criswell, and C. Ding, "Codestitcher: Inter-procedural basic block layout optimization," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019, Washington, DC, USA: Association for Computing Machinery, 2019, pp. 65–75. DOI: 10.1145/3302516.3307358.

[79] M. Chabbi, J. Lin, and R. Barik, "An experience with code-size optimization for production iOS mobile applications," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, US: IEEE Press, 2021, pp. 1–12. DOI: 10.1109/CGO51591.2021.9370306.

[80] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 378–415, Mar. 2000, ISSN: 0164-0925. DOI: 10.1145/349214.349233. [Online]. Available: https://doi.org/10.1145/349214.349233.

[81] L. Torczon and K. Cooper, *Engineering A Compiler*, 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, ISBN: 012088478X.

[82] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, Orlando, Florida, USA: ACM, 1994, pp. 159–170.

[83] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 181–210, Apr. 1991, ISSN: 0164-0925. DOI: 10.1145/103135.103136. [Online]. Available: https://doi.org/10.1145/103135.103136.

[84] T. J. Edler von Koch, B. Franke, P. Bhandarkar, and A. Dasgupta, "Exploiting function similarity for code size reduction," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14, New York, NY, USA: ACM, 2014, pp. 85–94. DOI: 10.1145/2666357.2597811.

[85] R. C. O. Rocha, P. Petoumenos, Z. Wang, M. Cole, K. Hazelwood, and H. Leather, "Hyfm: Function merging for free," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 110–121, ISBN: 9781450384728. DOI: 10.1145/3461648.3463852.

[86] *SimplifyCFG.cpp*, [Accessed 12-Apr-2023]. [Online]. Available: https://llvm.org/doxygen/SimplifyCFG_8cpp_source.html.

[87] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec. 2001, pp. 3–14.

[88] C. Saumya, K. Sundararajah, and M. Kulkarni, "Darm: Control-flow melding for simt thread divergence reduction," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '22, Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 28–40, ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741285. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741285.

[89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89, Austin, Texas, USA: ACM, 1989, pp. 25–35.

[90] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. Ferreira Guimarães, and F. M. Quinão Pereira, "Anghabench: A suite with one million compilable C benchmarks for code-size reduction," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 378–390. DOI: 10.1109/CGO51591.2021.9370322.

[91] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec. 2001, pp. 3–14.

[92] SPEC, *Standard Performance Evaluation Corp Benchmarks*, http://www.spec.org, 2014.

[93] *llvm-size - print size information*, [Accessed 13-April-2023]. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-size.html#:~:text=llvm%2Dsize%20is%20a%20tool,prints%20size%20information%20for%20a..

[94] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using peephole optimization on intermediate code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1, pp. 21–36, Jan. 1982. DOI: 10.1145/357153.357155.

[95] G. Lóki, Á. Kiss, J. Jász, and Á. Beszédes, "Code factoring in GCC," in *Proceedings of the 2004 GCC Developers' Summit*, 2004, pp. 79–84.

[96] M. Lika, "Optimizing large applications," *arXiv preprint arXiv:1403.6997*, 2014.

[97] S. Stirling, R. Rodrigo C. O., K. Hazelwood, H. Leather, M. OBoyle, and P. Petoumenos, "F3m: Fast focused function merging," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2022, pp. 242–253. DOI: 10.1109/CGO53902.2022.9741269.

[98] W.-K. Chen, B. Li, and R. Gupta, "Code compaction of matching single-entry multiple-exit regions," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, San Diego, CA, USA: Springer-Verlag, 2003, pp. 401–417, ISBN: 3540403256.

[99] S. Khurshid, C. S. Psreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'03, Warsaw, Poland: Springer-Verlag, 2003, pp. 553–568, ISBN: 3540008985.

[100] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224.

[101] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The opensmt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 150–153, ISBN: 978-3-642-12002-2.

[102] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[103] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223, ISBN: 1595930566. DOI: 10.1145/1065010.1065036. [Online]. Available: https://doi.org/10.1145/1065010.1065036.

[104] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Postconditioned symbolic execution," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102601.

[105] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," vol. 22, no. 4, Oct. 2013, ISSN: 1049-331X. DOI: 10.1145/2522920.2522925. [Online]. Available: https://doi.org/10.1145/2522920.2522925.

[106] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[107] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar, "Accelerating array constraints in symbolic execution," ser. ISSTA 2017, Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 68–78, ISBN: 9781450350761. DOI: 10.1145/3092703.3092728. [Online]. Available: https://doi.org/10.1145/3092703.3092728.

[108] H. Converse, O. Olivo, and S. Khurshid, "Non-semantics-preserving transformations for higher-coverage test generation using symbolic execution," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 241–252. DOI: 10.1109/ICST.2017.29.

[109] W. Chuang, B. Calder, and J. Ferrante, "Phi-predication for light-weight if-conversion," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003, pp. 179–190. DOI: 10.1109/CGO.2003.1191544.

[110] R. C. O. Rocha, C. Saumya, K. Sundararajah, P. Petoumenos, M. Kulkarni, and M. F. P. OBoyle, "Hybf: A hybrid branch fusion strategy for code size reduction," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023, Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 156–167, ISBN: 9798400700880. DOI: 10.1145/3578360.3580267. [Online]. Available: https://doi.org/10.1145/3578360.3580267.

[111] LLVM Compiler Infrastructure, *Llvm language reference manual*, https://llvm.org/docs/LangRef.html, [Accessed 23-Feb-2023], 2003.

[112] Free Software Foundation, *Variadic Functions (The GNU C Library) — gnu.org*, https://www.gnu.org/software/libc/manual/html_node/Variadic-Functions.html, [Accessed 23-Feb-2023], 2018.

[113] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982, ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(82)90008-6. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0196677482900086.

[114] D. Phillips, *Image Processing in C*. New Delhi, India: BPB Publications, 2008, ISBN: 9788170295150.

[115] I. Sobel, "An isotropic 3x3 image gradient operator," *Presentation at Stanford A.I. Project 1968*, Feb. 2014.

[116] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844.

[117] *The GNU oSIP library*, [Accessed 18-Apr-2023]. [Online]. Available: https://www.gnu.org/software/osip/.

[118] *GNU Libtasn1*, [Accessed 18-Apr-2023]. [Online]. Available: https://www.gnu.org/software/libtasn1/.

[119] *Coreutils - GNU core utilities*, [Accessed 18-Apr-2023]. [Online]. Available: https://www.gnu.org/software/coreutils/.

[120] *chcon(1) - Linux man page*, [Accessed 18-Apr-2023]. [Online]. Available: https://linux.die.net/man/1/chcon.

[121] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 350–360, ISBN: 9781450356381. DOI: 10.1145/3180155.3180251. [Online]. Available: https://doi.org/10.1145/3180155.3180251.

[122] *OSDI'08 Coreutils Experiments*, [Accessed 18-Apr-2023]. [Online]. Available: https://klee.github.io/docs/coreutils-experiments/.

[123] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344, ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. [Online]. Available: https://doi.org/10.1145/3133956.3134020.

[124] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpiski, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 1017–1032, ISBN: 9781450383912. DOI: 10.1145/3453483.3454092. [Online]. Available: https://doi.org/10.1145/3453483.3454092.

VITA

Charitha Saumya Gusthinna Waduge is a PhD candidate in the School of Electrical and Computer Engineering at Purdue University. He is advised by Prof. Milind Kulkarni. He received his B.S. in Electronic and Telecommunication from University of Moratuwa, Sri Lanka in 2015 and his M.S. in Electrical and Computer Engineering from Purdue University, West Lafayette, Indiana in 2020. He worked as a Software Engineer in Paraqum Technoligies, Sri Lanka from 2015 to 2016. He started his PhD in Electrical and Computer Engineering at Purdue University in August 2016. He was a software engineering intern at Bigstream Solutions in the summer of 2019. He was a research intern at Adobe Research in the summer of 2020. His current research is focused on developing targeted compiler transformations to improve performance and reliability of programs with irregular control-flow. He is passionate about developing compiler techniques to make software more performant, reliable and secure.