Control Flow Merging: A Compiler Transformation to Mitigate Branch Misprediction by Branch Elimination

Srinivasan Ramachandra Sharma

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Application

Kirshanthan Sundararajah, Chair
Xun Jian
Muhammad Ali Gulzar

June 23, 2025
Blacksburg, Virginia

Keywords: Compilers, Branchless Code, Branch misprediction

Control Flow Merging: A Compiler Transformation to Mitigate Branch Misprediction by Branch Elimination

Srinivasan Ramachandra Sharma

(ABSTRACT)

Modern superscalar processors have a high theoretical throughput, being capable of executing multiple instructions per clock cycle. In practice, however, this peak is rarely reached because data, structural, and control flow hazards can stall the pipeline and interrupt execution. Branch prediction is widely adopted in modern architectures to mitigate control flow hazards, yet even sophisticated predictors struggle to handle branches driven by random data. Each misprediction flushes speculative instructions and restarts execution on the correct path, making branch mispredictions a major performance bottleneck.

Existing compiler solutions, such as outcome pre-computation and simple predication, are effective only for loop-invariant conditions or small side-effect-free branches. This work introduces Control Flow Merging (CFM), a compiler transformation that eliminates branches by converting control dependencies into data dependencies. CFM systematically replaces if and if-then-else regions with semantically equivalent, predicated instruction sequences, even when handling operations with side effects. Our work targets branch elimination at the compiler level by converting control flow dependencies into dataflow.

We introduce the Control Flow Merging (CFM) pass for branch removal which replaces *if* and *if-then-else* regions with predicated instructions, while maintaining semantics and safety, even in the presence of instructions with side effects and implement it in LLVM 14. We also evaluate its impact on real-world benchmarks. Our experiments show that CFM can reduce branch mispredictions by up to 99 % and improve performance. Across the entire benchmark

set, the pass delivers up to 53x speedup in the best case, and a geometric-mean speedup of 1.57×.

Control Flow Merging: A Compiler Transformation to Mitigate Branch Misprediction by Branch Elimination

Srinivasan Ramachandra Sharma

(GENERAL AUDIENCE ABSTRACT)

Modern computer processors are built to perform many tasks at once to run programs faster. However, these processors often run into delays caused by having to guess the path a program will take next—especially when decisions depend on unpredictable data, like user input or random values. When the prediction is incorrect, the processor must discard the speculative work and restart from the correct path, leading to delays. While hardware improvements have tried to make these guesses more accurate, they still fall short in many real-world situations. Some programming tools also try to help, but they only work in very limited cases. This research introduces a new tool called Control Flow Merging (CFM), which helps the computer avoid making these guesses in the first place. Instead of guessing, CFM rewrites programs so that both possible paths can be handled safely without making a decision at all. This technique is added into a compiler commonly used for building programs and is tested on many common software workloads. The results show that CFM can significantly reduce delays and make programs run faster by up to 53 times faster in the best case. Overall, this work shows that smart programming tools can help computers make fewer mistakes and work more efficiently, even when working with unpredictable data.

Acknowledgments

A special thanks to my research advisor, Dr. Kirshanthan Sundararajah, for his invaluable guidance and mentorship throughout my graduate studies. I am also grateful to Dr. Charitra Saumya for his support and collaboration during the preparation of the research publication that forms a core part of this thesis. Their insights and encouragement have been instrumental in shaping this work.

Contents

List of Figures

List of Tables

List of Abbreviations

ALU Arithmetic Logic Unit

BB Basic Block

BPKI Branches Per Kilo Instruction

CFG Control Flow Graph

CFM Control Flow Merging

CPU Central Processing Unit

DT Dominator Tree

GEP Get Element Pointer

IPC Instructions Per Cycle

IR Intermediate Representation

LLVM Low Level Virtual Machine

MPKI Mispredictions Per Kilo Instruction

PDT Post-Dominator Tree

PEBS Precise Event-Based Sampling

SIMD Single Instruction, Multiple Data

Chapter 1

Introduction

1.1 Branch Misprediction

Superscalar processors are capable of executing multiple operations simultaneously during each clock cycle, offering the potential for extremely high throughput. In theory their performance is only bounded by the number of execution units or available SIMD lanes. In practice, however, real world workloads rarely reach this potential due to frequent disruptions in the instruction pipeline. These disruptions arise from various factors, including dependencies between instructions, memory access latency, and the presence of control-flow branches [21, 23]. To reduce the performance penalties introduced by branching, contemporary processors employ speculative execution driven by branch prediction—where the system temporarily follows a predicted path until the actual outcome of the decision point is known [18, 19, 34].

Although branch prediction is critical for keeping instruction pipelines busy, the cost of incorrect predictions remains substantial [14]. When a branch outcome is mispredicted, the processor must discard all speculatively executed instructions, reset the reorder buffer, and resume fetching from the correct program location [18, 40]. This corrective step introduces multi-cycle delays, often exceeding the depth of the frontend pipeline itself [8, 23, 26], ultimately wasting computation and reducing overall performance.

The performance cost of branch mispredictions has been well-documented in architectural research. Eyerman et al.[8] show that the latency introduced by incorrect predictions often exceeds the depth of the frontend pipeline, sometimes reaching 20–30 cycles due to secondary effects such as instruction window pressure and reorder buffer saturation. These stalls can significantly impair throughput in superscalar processors. Biggar et al.[3] further analyze this effect in the context of sorting algorithms, demonstrating that even small misprediction delays can substantially impact overall execution time. More recently, Lin and Tarsa [26] examined contemporary predictors and attributed as much as 18% of Instructions Per Cycle (IPC) loss directly to branch misprediction. Their findings suggest that eliminating mispredictions could yield performance improvements on par with an entire processor generation upgrade. Together, these studies underscore the limitations of hardware predictors and highlight the potential for compiler-based techniques to complement them by reducing the frequency and impact of branches in compiled code.

These penalties are particularly pronounced in the case of load-dependent branches, where the decision logic hinges on memory values that become available only at runtime [9, 10, 14]. Such branches are notoriously hard to predict and frequently appear in loops [12] and recursive constructs, amplifying their performance impact. In these contexts, branch mispredictions can dominate execution cost and represent a major obstacle to efficient out-of-order processing [3, 8, 26].

Several hardware-based solutions have been developed to reduce the performance impact of branch mispredictions. Among these, load/store-based predictors [1, 9, 10, 14, 15, 37] attempt to leverage correlations between memory access patterns and branch outcomes. While this approach can be effective under favorable conditions, it becomes unreliable when Load-Dependent Branches (LDBs) trigger cache misses, causing the processor to fetch and later discard instructions along the incorrect execution path [12].

Another class of techniques, including backsliding and continuous runahead execution [6, 16, 17, 29, 31, 35, 36], speculatively executes dependence chains in advance to precompute branch outcomes. Although this reduces misprediction latency, it demands additional hardware support and is less effective for loop-heavy workloads with short iteration counts, as time is required to learn and exploit the underlying data patterns.

Importantly, these hardware approaches attempt to enhance prediction accuracy rather than addressing the root cause—control-flow complexity. They retain conditional branches and merely seek to reduce their misprediction frequency, rather than eliminating them altogether.

Beyond hardware methods, compiler-level strategies have also been explored to mitigate branch mispredictions. An early example is the work by Kreahling et al. [22], which focused on scalar processors and aimed to reduce branching frequency by merging multiple conditional branches into a single branch and transforming control logic using bitwise operations. While effective in its historical context, the technique was evaluated on older architectures, and its applicability to modern superscalar CPUs remains uncertain due to substantial architectural changes over time.

A more recent effort, BOSS, proposed by Goudarzi et al. [12], decouples branch condition computation from the main execution path by precomputing outcomes ahead of loop iterations. This approach is particularly effective when branch conditions are loop invariant, but its scope is inherently limited to such cases.

Although both techniques aim to reduce misprediction overhead by altering control-flow structure, neither is capable of fully eliminating branches.

1.2 Control Flow Merging

This paper introduces Control Flow Merging (CFM), a compiler transformation that eliminates branches by converting control dependencies into data dependencies in structurally suitable control-flow regions. In doing so, CFM subsumes prior efforts like Kreahling et al.'s branch consolidation strategy, as it achieves similar outcomes while removing branches entirely within mergeable patterns, including those involving side effects.

CFM operates as a general-purpose compiler transformation that merges divergent execution paths in control-flow graphs, completely removing branches in structurally compatible regions. Its design surpasses earlier methods such as those proposed by Kreahling et al.[22], by enabling branch removal at the Intermediate Representation (IR) level while also supporting instructions with side effects. Compared to more recent efforts like BOSS[12], which rely on precomputing loop-invariant conditions, CFM enables broader applicability by handling branches whose conditions change dynamically across iterations, as seen in sorting and similar workloads.

The transformation begins by scanning the program's Control-Flow Graph (CFG) to locate `if` and `if-then-else` patterns that meet the criteria for merging. When such a region is identified, CFM aligns the divergent instructions across both paths. In cases where instructions differ, CFM introduces redundant but safe operations to achieve alignment. These added instructions do not alter program behavior, and instructions with side-effects are handled cautiously to prevent changes to program semantics. Special care is also taken to ensure that inserted operations involving memory accesses do not violate program safety.

Once alignment is achieved, the transformation rewrites the divergent logic into a unified sequence using predicated instructions—specifically, LLVM IR `select` operations, which compile down to conditional moves on architectures like x86. CFM aims to minimize the

use of `select` where possible by identifying equivalent or live operands across both paths. This process is repeated recursively until no additional mergeable regions remain, ultimately reducing or eliminating branches across entire functions. Unlike naive linearization strategies, CFM ensures minimal instruction overhead through careful operand management and alignment logic.

Our contributions include the design of a correct and semantics-preserving transformation for eliminating branches from mergeable control-flow regions, and its implementation as a pass in LLVM IR, facilitating integration with modern compilation pipelines. We conduct an empirical analysis using real-world benchmarks to quantify its impact. In addition, we investigate cases where CFM is less effective, such as memory-bound code or branches with already high prediction accuracy, highlighting the transformation's boundaries and offering direction for future refinement.

1.3 Motivating Example

To illustrate the performance implications of conditional branches, consider two function variants that convert lowercase letters to uppercase. The first uses a traditional conditional branch, while the second avoids branching entirely.

The function `to_upper_branchless` is functionally equivalent to `to_upper`, but differs in structure by removing the conditional branch entirely. Instead of relying on an `if` statement, it employs straight-line logic within the loop body. On x86 architectures, the terenary operator is typically translated to `cmov` instructions, which enable conditional assignment without branching. As a result, the branchless version exhibits virtually no branch mispredictions and achieves a substantial performance gain—up to 53× faster than its branch-based counterpart.

```c
char to_upper(char c) {
    if (c >= 'a' & c <= 'z')
        return c - 32;
    return c;
}

char to_upper_branchless(char c) {
    int cond = (c >= 'a') & (c <= 'z');
    return cond ? (c - 32) : c;
}
```

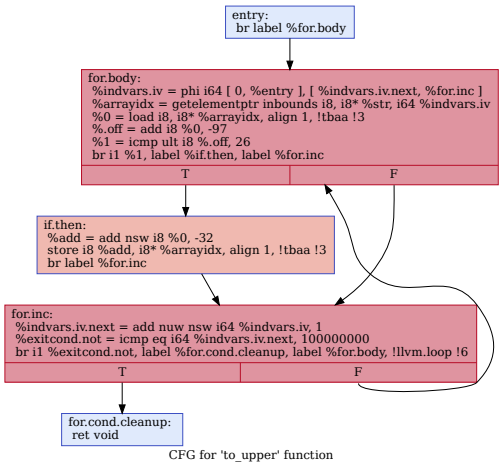
Listing 1: Motivating Example

In this version, the computation that adjusts the character value is executed unconditionally, regardless of whether the condition originally guarding it would have been true or false. When the condition is not met, a zero offset is applied, which leaves the character unchanged. This preserves program correctness while eliminating speculative execution entirely.

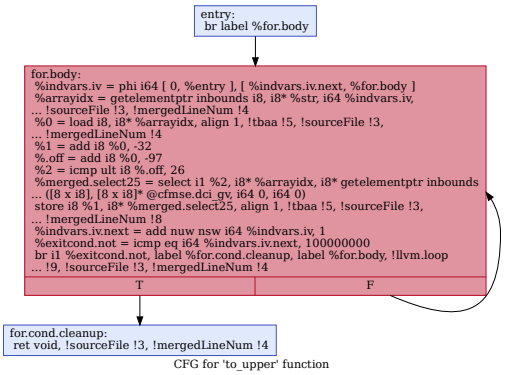
This example highlights the central insight behind the CFM transformation, inserting redundant but safe instructions to unify divergent control-flow paths and avoid explicit branching. Beyond reducing misprediction penalties, eliminating branches facilitates aggressive compiler optimizations such as vectorization. Even when vectorization is disabled, the branchless version achieves a 10× speedup and reaches an IPC of 6.23—demonstrating the practical benefits of branch elimination in modern CPU architectures.

While this transformation can be performed at the source code level, it can also be applied more systematically at the Intermediate Representation (IR) level. Figure 1.1a shows the LLVM IR corresponding to the original `to_upper` function, while Figure 3.3b illustrates the IR after applying the CFM transformation.

In the following sections, we detail the design and safety considerations of the CFM transformation and present an experimental evaluation that explores its strengths and limitations.



(a) Original control-flow of `to_upper` function



(b) Merged control-flow of `to_upper` function

Figure 1.1: Merging control-flow at the LLVM-IR level

Chapter 2

Literature review

2.1 Hardware Techniques

Numerous architectural studies have focused on enhancing branch prediction mechanisms to reduce pipeline stalls. Traditional history-based predictors have been extended and refined over time to improve accuracy [27]. In parallel, load/store-based predictors have emerged as a promising direction, correlating memory access patterns with branch behavior to improve prediction outcomes [1, 9, 10, 15, 37]. More recently, machine learning techniques have gained traction in this space, with perceptron-based models [19, 20], neural networks [39], and alternative architectures such as WiSARD classifiers [38] and bit-level perceptron variants for indirect branches [11], all contributing to more adaptive and data-driven prediction strategies.

Beyond prediction itself, alternative mechanisms such as Branch Runahead [17, 29, 31] have been proposed. These techniques aim to bypass prediction altogether by speculatively executing short dependence chains leading to the branch condition, allowing the outcome to be resolved earlier in the pipeline and avoiding mispredictions entirely.

2.2 Software Techniques and Algorithms

Software-level approaches have also been explored extensively to reduce the impact of branch mispredictions, particularly in workloads dominated by control-flow decisions. Gao et al. [13] propose branchless algorithms for graph traversal, specifically targeting the computation= graph algorithms. Despite introducing additional memory operations, their approach yields notable performance improvements by minimizing branch mispredictions. Similarly, Elmasry and Katajainen [7] show that rewriting traditional comparison-based search procedures into branchless equivalents can significantly lower misprediction rates and improve execution speed. Langdale and Lemire [24] take this principle further in the context of data parsing, introducing `simdjson`, a high-throughput JSON parser that employs branch-avoiding strategies to enhance vectorization efficiency, achieving multi-gigabyte-per-second performance per core.

2.3 Compiler Optimizations

Compiler-driven transformations to reshape control flow have be studied across diverse domains. In symbolic execution, control-flow simplification helps mitigate path explosion by reducing the number of branches that must be explored when checking for bugs in code [33]. In vectorization, control-flow restructuring allows loops with divergent paths to be rewritten into uniform sequences, enabling efficient execution through predication or conditional operations [4, 28, 30]. Similarly, in GPU programming, compiler optimizations have been employed to reduce Single Instruction, Multiple Thread (SIMT) divergence, which arises when different threads within an execution group follow different execution paths [2, 5**? ?**]. While these efforts primarily target symbolic reasoning, SIMD efficiency, or GPU exe-

cution models, the underlying control-flow manipulation strategies are directly relevant to mitigating branch mispredictions on CPUs.

Recent compiler research has explored techniques explicitly designed to reduce such mispredictions. Goudarzi et al. [12] propose BOSS, a system that precomputes branch conditions within loops by extracting minimal dependence slices (backslices) required to evaluate each condition. These backslices are speculatively executed before the main loop, enabling the branch outcomes to be cached and reused during loop execution. This reduces mispredictions but is inherently limited to loop-invariant or statically analyzable conditions. In contrast, CFM targets a broader class of branches by structurally merging divergent regions directly in the compiler's intermediate representation, regardless of whether the condition varies dynamically. This makes CFM especially effective in applications such as sorting, where traditional precomputation techniques like BOSS fall short. BOSS is orthogonal to our work and can potentially be combined to further reduce branch-related stalls.

2.3.1 Existing LLVM Passes and Limitations

LLVM includes several optimization passes capable of merging control flow by replacing conditional branches with dataflow constructs through the use of select instructions. These include InstCombine, SROA, Branch Fusion, SimplifyCFG, and Function Merging. While each of these passes serves a broader purpose—ranging from improving runtime performance and reducing code size to enabling later-stage optimizations—their ability to eliminate branches is especially relevant to the problem statement i.e mitigating branch mispredictions.

At first glance, these built-in transformations may seem sufficient for producing branchless code automatically. However, their branch removal capabilities are often limited to narrow patterns, such as constant assignments or simple arithmetic. In practice, each pass imposes

strict structural or semantic constraints that restrict their application. These limitations, and how they compare to Control Flow Merging (CFM), are explored in the sections that follow.

2.3.2 InstCombine Pass

The InstCombine pass in LLVM is a low-level, peephole-style optimization pass that focuses on simplifying individual instruction sequences within basic blocks. Its primary objective is to reduce redundancy and replace arithmetic, logical, and bitwise operations into more efficient or semantically cleaner forms. This pass operates locally and does not perform control-flow restructuring in the general case. Instead, it targets sequences of instructions that can be simplified or folded, such as constant expressions, repeated operations, strength reductions (e.g., replacing multiplications with shifts etc.), or redundant type conversions.

InstCombine also manages SSA constructs like phi nodes, particularly when the incoming values are trivially computable or equivalent. In such cases, it may introduce a select instruction to replace a phi, provided that the condition controlling the value choice is already available and the values can be safely evaluated on all paths.

In some instances, InstCombine can partially simplify the result of branching logic by converting only specific SSA variables into select-based expressions, while leaving the surrounding control structure unchanged. For example, if one path of a conditional assigns a constant and the other performs a simple arithmetic operation, the resulting phi node may be replaced by a select, even though the branch itself remains in the control flow graph as shown in Figure 2.1

```
entry:
  %cond = icmp eq i32 %x, 0
  br i1 %cond, label %then, label %else

then:
  %add_then = add i32 %x, 1
  %mul_then = mul i32 %x, 42
  br label %merge

else:
  %add_else = add i32 %x, 2
  %sub_else = sub i32 %x, 5
  br label %merge

merge:
  %val = phi i32 [ %add_then, %then ],
                 [ %add_else, %else ]
  ...
  ...
```

(a) Original code

```
entry:
  %cond = icmp eq i32 %x, 0
  br i1 %cond, label %then, label %else

then:
  %mul_then = mul i32 %x, 42
  br label %merge

else:
  %sub_else = sub i32 %x, 5
  br label %merge

merge:
  %sel = select i1 %cond, i32 1, i32 2
  %val = add i32 %x, %sel
  ...
  ...
```

(b) Transformed using `select` instruction

Figure 2.1: InstCombine can remove SSA phi nodes by using `select` instructions.

2.3.3 Scalar Replacement of Aggregates Pass

The Scalar Replacement of Aggregates (SROA) pass in LLVM is an early memory-to-register promotion pass. Its goal is to simplify memory access to composite types (structs or arrays), by breaking them down into individual scalar variables. This transformation enables later passes to operate on values in SSA form instead of performing repeated memory loads and stores.

SROA is not a control-flow optimization pass, however it plays an indirect role in enabling control-flow merging. By lifting memory accesses into registers, SROA allows conditional assignments to be expressed as SSA values. This, in turn, can make patterns involving conditional stores or loads more amenable to simplification by other passes like InstCombine or SimplifyCFG.

In some cases, this can lead to branch removal, as subsequent passes may recognize a conditional store pattern and rewrite it as a select between scalar values. However, SROA itself does not modify control flow directly.

2.3.4 Function merging

The function merging pass in LLVM is designed to reduce code size by combining entire functions that are structurally similar. It identifies functions with nearly identical sequences of instructions and merges them into a single implementation, by introducing a dispatch mechanism that selects the correct behavior at runtime.

When two functions are merged, the compiler can introduce either introduce new control flow or data-selection logic to reconcile differences in behavior. While this preserves functional correctness, it does not always reduce branching and can introduce new conditional branches where none existed previously. Furthermore, the pass is conservative in its application: small differences in structure or control flow often prevent merging, and the transformation is all-or-nothing at the function level. Due to these characteristics, function merging is not well suited for mitigating branch mispredictions.

2.3.5 Branch Fusion

Branch Fusion in LLVM attempts to merge similar basic blocks within the same function, aiming to reduce code size by combining duplicated logic. Like Control Flow Merging (CFM), it identifies branches with matching structures and aligns their blocks. However, its primary goal is code size reduction, not performance optimization. The pass uses block merging functions provided by the function merging pass, as suffers from similar limitations, which reduce its ability to remove branch mispredictions.

2.3.6 Simplify CFG

The SimplifyCFG pass in LLVM performs simplification of the control-flow graph (CFG) in order to reduce branching overhead and prepare the intermediate representation for further optimization. This pass encompasses a wide range of transformations, including the elimination of empty basic blocks, merging of blocks with single predecessors or successors, and simplification of redundant phi nodes. Crucially, it also includes logic for converting certain conditional branches into dataflow operations, particularly through the use of the select instruction. This transformation replaces branches with value selection logic when conditions permit, effectively reducing the control-flow complexity of the IR.

One of the key operations SimplifyCFG performs is branch elimination via value merging. When an if-else structure contains symmetric assignments to the same variable or memory location, and the control paths reconverge immediately after, SimplifyCFG may eliminate the branch and use a select to compute the result unconditionally. This transformation is only applied when both control paths assign to the same target and involve values that are safe to compute outside of their original control-flow context—that is, they must be free of side effects and memory hazards.

```
if (cond)
    a[i] = a[i] * 5 + 32;
else
    a[i] = 5;
```

(a) Code Mergable by Simplify CFG

```
if (cond)
    a[i] = a[i] * 5 + 32;
else
    a[i + 1] = 5;
```

(b) Code Unmergable by Simplify CFG

Figure 2.2: examples of Code mergable by SimplifyCFG.

Figure 2.2a and Figure 2.2b show two conditionals where SimplifyCFG behaves differently. In the first case, both branches store to the same memory location a[i], allowing the branch to be replaced with a select on the value. In the second case, the store targets differ—a[i] vs.

a[i+1]—which prevents merging. SimplifyCFG can only select between values, not addresses, so the second pattern requires retaining the conditional branch.

One-sided conditionals, like the example shown in Figure 2.3, are common in real-world programs. These structures apply a modification only under specific conditions, leaving the variable unchanged otherwise. However, because there is no well-defined value assigned on the false path, SimplifyCFG cannot form a valid phi node (phi node is not required due to memory store) or select instruction. This makes such branches ineligible for merging. Despite their simplicity, one-sided branches pose the same performance risks as two-sided ones in terms of misprediction, yet LLVM cannot eliminate them using this pass.

```
if (cond)
    a[i] = a[i] * 5 + 32;
else
    a[i + 1] = 5;
```

Figure 2.3: SimplifyCFG cannot merge one-sided conditionals due to the absence of a defined value on the false path.

2.3.7 Pass Coverage and Gaps

In conclusion, LLVM provides multiple optimization passes—such as InstCombine, SimplifyCFG, Branch Fusion, and Function Merging—that are capable of merging basic blocks, simplifying control flow, and replacing conditional branches with select instructions where possible. While these transformations are effective in reducing code size and improving IR regularity, they are conservative by design and often fail to eliminate branches in structurally divergent cases. Common patterns such as one-sided conditionals or differing memory access paths are left unmerged, limiting the effectiveness of these passes in mitigating branch

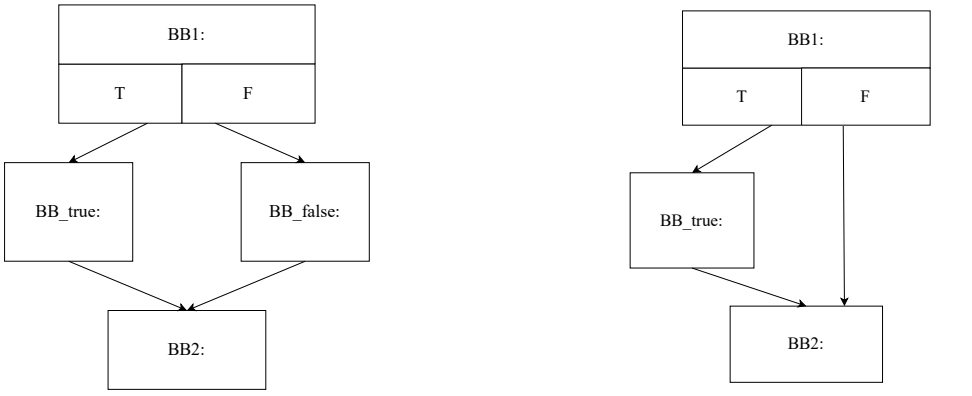
misprediction penalties. Notably, all the examples discussed in this section—including those that LLVM fails to simplify—can be merged by the Control Flow Merging (CFM) transformation proposed in this thesis.

Chapter 3

Design

3.1 Definitions

We refer to the following definitions throughout this section and in subsequent discussions.



(a) Diamond Control Flow (b) Diamondoid Control Flow

Figure 3.1: Mergable Control Flow

Definition 3.1 (Diamond Control Flow)**.** A control-flow region is *diamond-shaped* (Figure 3.1a) if it consists of a conditional branch from a single basic block (BB) where:

- Both branch targets are distinct single entry, single exit BBs.
- Both BBs reconverge at the same point.

Definition 3.2 (Diamondoid Control Flow)**.** A control-flow region is *diamondoid-shaped* (Figure 3.1b) if it consists of a conditional branch from a single basic block (BB) where:

- One branch target is a single BB with a single entry (from the branch) and single exit (reconvergence point).
- The other branch target is the reconvergence point itself.

Definition 3.3 (Merged Basic Block)**.** A *merged basic block* is the resulting basic block produced by applying the control-flow merging transformation to a mergeable divergent branch.

3.2 Overview of the Transformation

Control-Flow Merging (CFM) is a general-purpose compiler transformation designed to operate at the intermediate representation (IR) level. It focuses on simplifying control structures by eliminating branches in diamond and diamondoid regions, and replacing them with a single basic block. These patterns represent common branching structures where execution diverges and reconverges, and they are ideal targets for restructuring. CFM replaces these control constructs with linear code that uses predicated execution to preserve correctness, thereby improving performance by reducing branch misprediction.

The transformation consists of 3 key steps namely the mergability check, instruction alignment and neutral code insertion. It begins by scanning the CFG to locate regions where control diverges and reconverges in a manner compatible with merging. Once identified, these candidate regions undergo structural validation to ensure that they meet the criteria for transformation. Valid regions are then prepared for merging by aligning their basic blocks. When mismatches occur between blocks, the transformation inserts non-disruptive

placeholder instructions to align the control paths. The operands for these added instructions are carefully managed to ensure they do not introduce unintended side effects.

In the final stage, the original branches are eliminated and replaced with a sequence of predicated operations using `select` instructions. These operations conditionally choose operands based on the original branch condition, effectively emulating the behavior of the branch without introducing control divergence. The result is a single, linear basic block that maintains the original program behavior while reducing the likelihood of misprediction.

3.2.1 One Sided Branch Canonicalization

The first step in the CFM transformation is to convert one-sided conditional branches, referred to diamondoid patterns, into two-sided (diamond-shaped) branches. This is done by inserting an empty basic block into the fall-through path, effectively standardizing the control-flow structure for further analysis and transformation. Figure 3.2 illustrates this process. This canonicalization is reguired for subsequent stages of instruction alignment and merging required by CFM.

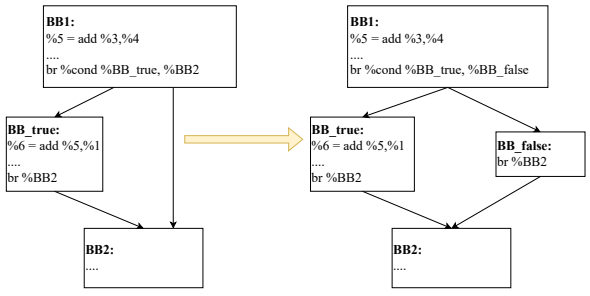


Figure 3.2: Processing One-sided branches to diamond

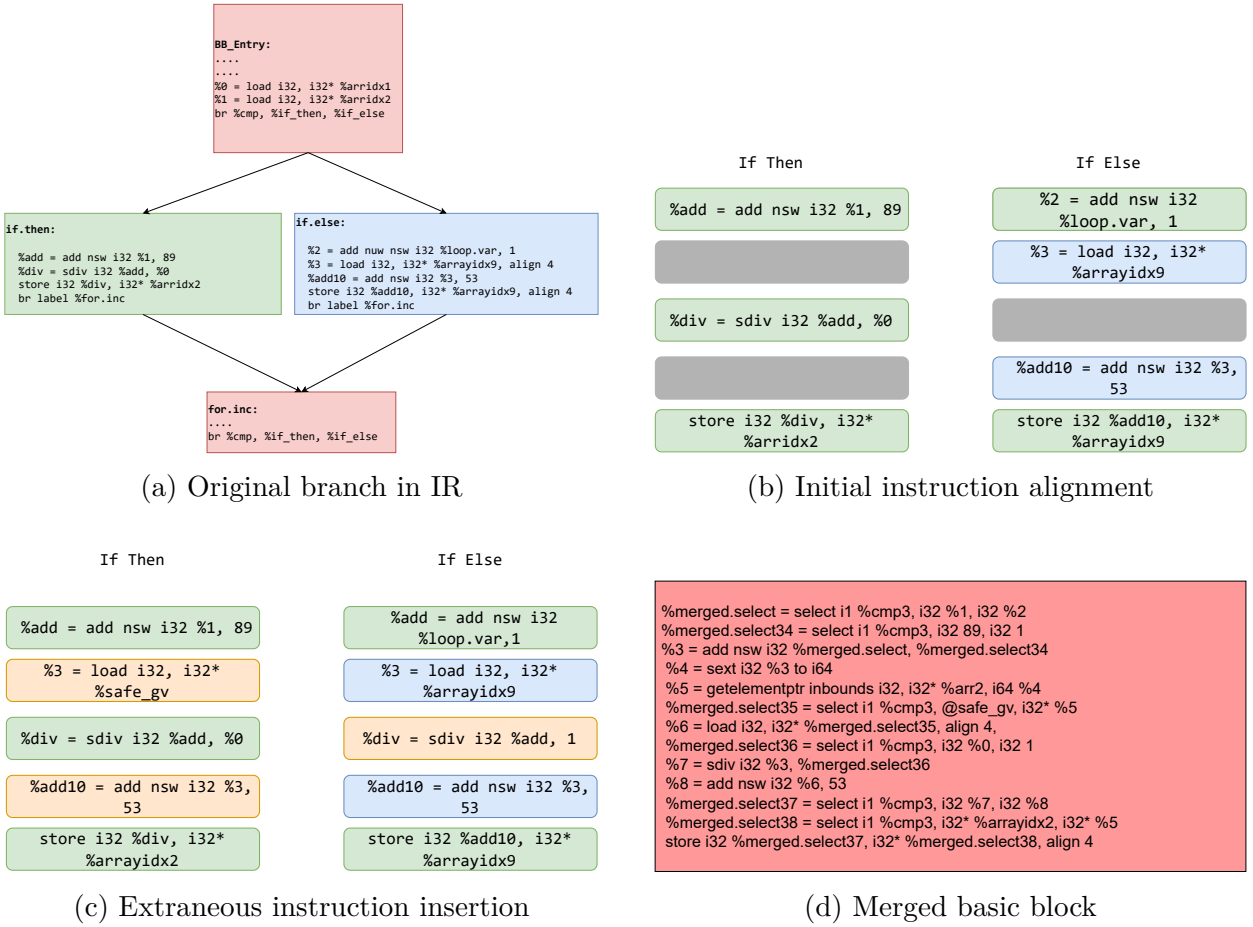
Meldability Check

Once conditional branches are detected, the next step in the CFM process is to assess whether they are eligible for merging. This decision is based on the algorithm outlined in Figure 3.5, which enforces a set of structural constraints to verify that the control-flow region adheres to the diamond-shaped pattern necessary for safe transformation.

The initial requirement is that the terminator of the basic block containing the branch must be a conditional branch instruction. Next it must dominate both of its target blocks, guaranteeing that any execution reaching either branch path must first pass through the branching block. The algorithm also disqualifies regions where direct control-flow edges exist between the two successor blocks, as such connections violate the separation needed between paths. Additionally, it confirms that neither successor post-dominates the other; if execution through one path always leads to the other, we cannot guarantee that only one of the basic blocks is run during execution. The separation between the successor basic blocks is extremely important, since the instruction alignment algorithm aims to align both paths and run only one conditionally.

The algorithm also filters out branches that include operations incompatible with CFM's merging logic. In particular, instructions such as function calls and vectorized memory accesses are excluded. Function calls often alter control flow or rely on external side effects that are difficult to replicate safely in both paths. Further, the performance cost of function calls is extremely high, and adding a function call to one of the paths could cause significant performance degradation. Similarly, vector memory operations pose alignment and semantic challenges. Since CFM targets scalar IR before vectorization, the sophisticated handling required for vector loads and stores can be avoided. To avoid this complexity and ensure correctness, these instructions are not supported at the transformation phase.

These structural and semantic filters collectively ensure that CFM is applied only side effect free control-flow regions where merging can proceed safely and reliably.



(a) Original branch in IR

(b) Initial instruction alignment

(c) Extraneous instruction insertion

(d) Merged basic block

Figure 3.3: Instruction alignment and basic block merging

Alignment

Once a region is deemed suitable for merging, CFM proceeds by aligning instructions from the two diverging control paths this allows for neutral code insertion and merging using `select` operations (realized as conditional moves in X86 backend code geneartion). This alignment and merging process is illustrated in Figure 3.3, which transforms the branching structure shown in Figure 3.3a.

Instructions from both the true and false successor block are aligned sequentially in this step, if a similar instruction exists on both paths, they are placed side by side, if not a blank slot is inserted on one side. Let I_{true} and I_{false} denote the instructions along the two paths. The algorithm compares the instruction streams sequentially. If the current instructions on both sides share the same opcode and operand types (e.g., both are `add`, `mul`, or memory operations of the same kind), they are matched and considered aligned. When a mismatch occurs, the algorithm introduces a blank slot (a placeholder) on one side to maintain structural consistency. This alignment continues until all instructions from both paths are accounted for. An example of this preliminary alignment step is depicted in Figure 3.3b.

Limitation of Simple Instruction alignment

Instruction alignment is not a new concept and has been explored in prior work, such as DARM [32], particularly in the context of reducing thread divergence in GPUs. However, when applied to CPU programs, instruction alignment alone proves insufficient. As illustrated in Figure 3.3, merely aligning instructions without additional mechanisms like neutral code insertion results in unpredication of unaligned operations to preserve correctness. This process involves splitting control flow and introducing new branches, as seen in 3.3. The resulting increase in control-flow complexity can negate the benefits of alignment by inflating the total number of branches, thereby increasing the likelihood of branch mispredictions and potentially degrading overall performance.

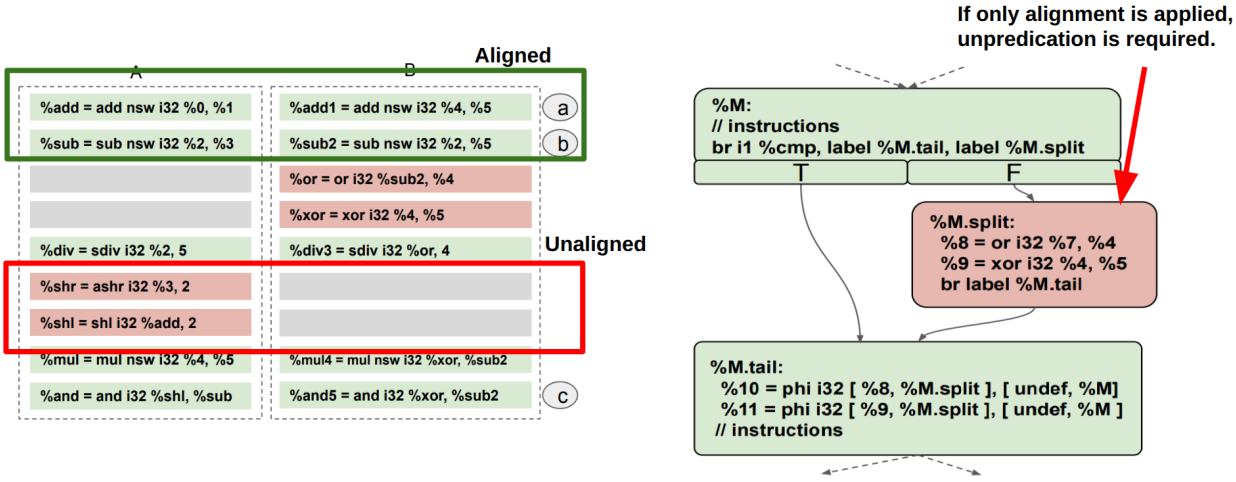


Figure 3.4: Limitation of just instruction alignment. Image courtesy of DARM [32].

Neutral Code Insertion

After completing instruction alignment, the transformation proceeds to choose appropriate, semantically neutral instructions to replace the blank spaces introduced by alignment. This helps in operand selection. This phase ensures that any inserted instructions behave safely and do not alter the intended semantics of the original program, allowing for block merging in the next step. Particular attention is required when dealing with unaligned instructions that can raise exceptions or access memory.

For instance, division operations may raise errors such as divide-by-zero if introduced without care. To avoid such issues, neutral divide instructions are inserted using a constant divisor of 1, ensuring that the operation remains valid regardless of which execution path is active.

A similar strategy applies to memory operations. Unconditional execution of memory loads or stores from a previously inactive path could result in unintended behavior (corrupting data) or program crashes. To safeguard against this, CFM replaces the address operand of such inserted memory instructions with a reference to a globally defined *safe pointer*. This

Algorithm 1: Check for Valid Merge Location

Input: Basic block BB, Dominator Tree DT, Post-Dominator Tree PDT
Output: **bool** indicating if BB is a valid merge location

```
1  Br ← terminator instruction of BB ;
2  if Br is NULL or number of successors of Br < 2 then
3      return false
4  BB_L ← first successor of Br ;
5  BB_R ← second successor of Br ;
6  if not Dominates(BB, BB_L) or not Dominates(BB, BB_R) then
7      return false
8  if Path exists from BB_L to BB_R then
9      return false
10 if not PostDominates(BB, BB_L) or not PostDominates(BB, BB_R) then
11     return false
12 if BB_L or BB_R contains unhandled instructions then
13     return false
14 return true
```

Figure 3.5: Algorithm to check if a basic block is a valid merge location.

pointer designates a harmless memory location, ensuring that any speculative access will not affect program state. By doing so, CFM preserves correctness while enabling predicated execution for all instructions in the divergent region.

Although the use of a safe pointer enables correct handling of both memory loads and stores, store instructions offer further room for optimization. Typically, choosing between memory addresses using conditional selection can add overhead. However, if the address being stored to was already loaded earlier in the control path while computing the branch condition, and remains unchanged, this overhead can be avoided.

For example, as illustrated in Figure 3.3a, if a value such as `arr2[i]` is loaded for a branch condition and remains untouched thereafter, it does not need to be loaded again during alignment. Instead, the transformation can reuse this preloaded value. In these scenarios,

Algorithm 2: Control Flow Merging Pass

Input: Basic block BB, dominator tree DT, post-dominator tree PDT, target transform info TTI

Output: `local_change` – true if merges occurred

```
; // Create internal cfm.safe_gv
1 cfm.safe_gv ← new internal 64-bit array;
; // Insert pointer to cfm.safe_gv into every BB
2 foreach B ∈ Function(BB) do
3     Insert GEP of cfm.safe_gv at entry of B;
4 Changed ← true;
5 while Changed do
6     Changed ← false
7     foreach F ∈ M do
8         processOneSidedBranches(F);
9         if runCFMOnFunction(F) then
10            Changed ← true;
```

Figure 3.6: Control Flow Merging Pass

rather than conditionally selecting a memory address, CFM conditionally selects the value to be stored. The store instruction is then constructed using the address of the original store and the value to be stored is selected. This effectively simplifies memory access and improves performance.

To ensure safety, the implementation applies a conservative approximation that eliminates only those redundant loads that are provably safe. Specifically, it discards a redundant load only if the memory location was previously loaded and no store instructions targeting any location appear before the merge point. We need to ensure that there are no loads after stores, to avoid issues with pointer aliasing.

Instruction alignment in CFM plays a critical role not only in ensuring correctness but also in maximizing performance. During merging, instructions from divergent paths must be aligned such that semantically equivalent operations execute in parallel. To preserve safety,

Algorithm 3: Run CFM on Function

Input: Function F, Analysis Manager AM, Options O
Output: Boolean `Changed` indicating whether modifications occurred

```
1  Initialize dominator tree DT, post-dominator tree PDT, loop info LI from F;
2  Obtain target transform info TTI from AM;
   ; // Collect brnaches based on compiler flags
3  ValidBranches ← ∅;
4  foreach basic block B ∈ F do
5      if B ends in a conditional branch BI then
6          if O.OnlyInLoops is true and B is not in a loop then
7              continue;
8          Add BI to ValidBranches;
   ; // Run Control FLow Merging on Region
9  Changed ← false;
10 foreach branch instruction BI ∈ ValidBranches do
11     if runCFM(Block, DT, PDT, TTI) then
12         Recalculate DT and PDT;
13         Changed ← true;
14 if Changed then
15     removeRedundantJumps(modified blocks);
16 return Changed;
```

Figure 3.7: Algorithm to apply CFM transformation to all valid conditional branches in a function.

the transformation introduces neutral or redundant operations where necessary. However, excessive use of select instructions to resolve operand differences can hurt performance, especially when they introduce unnecessary dependencies on condition evaluation. To mitigate this, the alignment phase aggressively reuses operands that are live and valid across all control-flow paths. These operands can be directly propagated without wrapping them in select instructions, enabling earlier execution and reducing pipeline stalls. When both operands are live on both paths this optimization minimizes conditional logic overhead as well, contributing to faster execution in the merged code.

Consider Figure 3.3c, where an `add` instruction exists solely on the true path. Since its

operands are live on both paths and executing the operation poses no safety risks, it can be directly added to the unified block without a `select`. By contrast, the subsequent load instruction necessitates full operand selection to ensure semantic safety. The location of the load is conditionally selected, with either arr2[i+1] or the safe_gv. The final example, a division operation, illustrates a hybrid case: only the second operand is subject to selection to avoid potential divide-by-zero faults, as the first operand is valid on both paths.

3.2.2 Merging

After neutral code insertion, the transformation proceeds to introduce `select` instructions that determine the appropriate value from each control-flow path for use in the merged block. The neutral code insertion phase applies strategies to reduce the number of `select` instructions whenever possible by reusing existing operands during the alignment stage, when this reuse preserves correctness and avoids unsafe behavior. The basic block merging performed by CFM is visualized in Figure 3.3d, where aligned instructions from the divergent control paths are unified, and differing operands are resolved using `select` instructions to preserve correctness while eliminating branches.

Elimination of Redundant Jumps and ϕ-Nodes

Once the CFM transformation has completed, the ϕ nodes in the post-dominating basic block of the transformed region (e.g., `BB2` in the running example) are no longer necessary. These nodes originally resolved values based on divergent paths; however, since control flow has been merged and decisions are now made via `select` instructions, the role of ϕ nodes is obsolete. Therefore, they are removed to simplify the IR.

Additionally, the transformation may introduce extra jump instructions—particularly from

the process of converting one-sided branches into standard two-way branches. These superfluous jumps, along with now-redundant basic blocks, are pruned during a final simplification phase, which restores a clean and efficient control-flow structure, as seen in 3.7

3.2.3 Compiler Integration

The CFM transformation has been developed as a dedicated pass within LLVM 14. It is integrated into the compilation pipeline and becomes active automatically at optimization levels above `-O0`. To ensure that the transformation is applied to the most canonical form of the code, CFM is positioned after initial IR simplification passes such as `SimplifyCFG`, `SROA`, and `EarlyCSE`. This ordering maximizes opportunities for merging and minimizes unnecessary duplication.

Being placed before target-specific backend optimizations ensures that the transformation does not interfere with analyses such as alias resolution or impact code generation decisions related to memory layout. By simplifying control flow early, CFM also enhances the effectiveness of later optimizations such as vectorization, leading to performance benefits that enhance the gains from eliminating branches.

Chapter 4

Evaluation

4.1 Benchmark Suite

To evaluate the effectiveness of the CFM transformation, we employ a benchmark suite composed of widely studied algorithms known to contain data-dependent branches [3, 13]. Each benchmark is executed using randomly generated inputs to simulate a wide variety of runtime behaviors. Input sizes are carefully selected based on recommendations from prior work [25] to ensure that the effects of branch misprediction are both measurable and significant.

The suite includes algorithms from multiple domains. In the category of array and set processing, we include `arrayIntersection`, `arrayMerge`, and `arrayUnion`, which operate on sorted array inputs, as well as `toUpper`, a simple case involving string manipulation. The sorting class features a broad set of techniques, including `bitonic`, `bubbleSort`, `heapSort`, `merge`, `qsort`, `shakerSort`, `shellSort`, `dutchFlag`, and `kadane`, providing coverage of both classical and advanced sorting and sequence analysis algorithms.

In the graph domain, we evaluate `ccomp` (connected components), `karger` (minimum cut), and both `kruskal` and `prim` (minimum spanning tree algorithms). The suite also includes two morphological image processing routines, `dilation` and `erosion`, and one simulation workload, `lbm`, which applies the Lattice Boltzmann method to fluid dynamics.

4.1.1 Experimental Setup

The Control Flow Merging (CFM) transformation is implemented as a custom pass at the LLVM IR level within the LLVM 14 infrastructure. All evaluations are conducted using the Clang compiler with the `-O3` optimization level enabled. For comparison, the baseline corresponds to the same `-O3` configuration with CFM explicitly disabled.

Experiments are executed on a machine equipped with an Intel(R) Xeon(R) Gold 6430 CPU, which offers 32 physical cores and supports 2 hardware threads per core. The system is provisioned with 256 GB of RAM and runs Ubuntu 22.04.5 LTS (Jammy Jellyfish).

This setup is used to investigate the performance impact and safety of the CFM transformation. The evaluation aims to answer the following research questions:

Branch Miss Reduction Potential (RQ1) : What is the upper bound on the percentage of branch mispredictions that can be eliminated by applying the CFM transformation (for the selected programs)?

Performance Across Programs (RQ2) : How effectively does the CFM transformation reduce branch mispredictions and improve the Instructions Per Cycle (IPC) in a program, and how does this contribute to faster execution times for applications running on a CPU?

Application Profitability (RQ3) : Under what conditions does CFM hurt performance and why?

4.1.2 Upper bound Analysis

To determine the upper bound of branch misses that can be eliminated with Control-Flow Melding (CFM), we use both compiler and performance analysis tools. First, we compile each program with debugging flags enabled (`-g`), which provides essential information for line mapping. Then, we use Intel's Precise Event-Based Sampling (PEBS) events via `perf` to obtain a highly accurate count of branches, branch misses, and total instructions executed. Then `perf script`, along with `addr2line`, can be used to map performance events to specific line numbers in the C code, enabling precise localization of branch-related metrics. We then manually review these lines to determine which are relevant to the core functionality of the program, filtering out library functions and initialization branches, which we categorize separately. This ensures that the data shown in the first four columns of Table 4.1 reflect only the branch and miss rates pertinent to the program under evaluation.

Finally, we can manually identify specific lines within the main program where CFM can be applied, enabling us to accurately determine the upper bound metrics for removable branch misses. We aggregate this information using a Python script to calculate the theoretical upper bound of removable branch misses. This process results in the data seen in Table 4.1.

To properly interpret the results, it is important to recall that CFM is applicable only to branches that satisfy specific structural constraints, namely diamond and diamondoid control-flow regions that do not contain function calls.

Certain benchmarks, including `dilation`, `erosion`, and `kadane`, already experience low branch misprediction rates under Clang's `-O3` optimizations, due to effective built-in branch elimination techniques (Figure 4.1). In contrast, `mergeSort` incurs mispredictions primarily from recursive calls guarded by conditional branches, constructs that fall outside the current capabilities of CFM and cannot be merged. Likewise, `shellSort` demonstrates elevated

misprediction rates stemming from control-flow within loop headers, which are not merged by CFM due to its structural limitations.

Figure 4.1: CFG of the body of the for loop in dilation (Compiled with -O3 cfm not enabled)

4.1.3 Miss Reduction and IPC increase

Our experimental results demonstrate that the normalized misses in the merged code are generally very close to the theoretical expectations, as shown in Table 4.2. In some cases, the experimental misses are even lower than the theoretical upper bound. For instance, benchmarks like `array-intersect`, `array-merge`, and `bubble-sort` achieve experimental miss rates that perfectly match or outperform the theoretical expectations. However, there are instances where the experimental results deviate from the theoretical predictions. Notable cases include `heap-sort` and `prim`, each of which presents unique characteristics worth further investigation.

Let us first consider the case of `heap-sort`. A key part of the algorithm is shown in Code 2:

As shown above, in `heap-sort`, there are three `if` conditions. The first two conditions are

Table 4.1: An empirical upper bound of the percentage of misses removable by CFM

Program	% Cond Br	% Cond Br Rmv.	% Cond Misses	% Miss Rmv.	BPKI	MPKI
arrayMerge	89.98	45.05	50.13	100.00	146.16	21.49
arrayUnion	89.19	45.87	50.13	100.00	136.32	19.90
arrayIntersection	100.00	43.04	50.27	100.00	167.40	24.45
bitonic	74.11	41.41	50.51	70.53	71.44	1.96
bubbleSort	75.30	67.21	50.20	100.00	100.52	17.59
ccomp	97.30	74.24	50.11	100.00	188.81	47.18
dilation	100.00	0.00	0.00	0.00	15.76	0.00
dutchFlag	76.56	75.14	50.14	100.00	141.26	44.42
erosion	100.00	0.00	0.00	0.00	9.96	0.00
heapSort	84.20	78.69	50.15	100.00	82.57	8.36
kadane	50.12	0.00	0.00	0.00	35.90	0.00
karger	92.96	47.40	37.94	50.00	177.15	3.38
kruskal	15.35	0.00	0.00	0.00	23.03	0.00
lbm	77.82	0.00	0.00	0.00	3.61	0.00
mergeSort	84.86	0.00	42.36	0.00	45.32	0.47
prim	95.00	51.80	50.22	100.00	158.01	18.61
qsort	93.88	64.85	49.88	95.32	132.38	8.99
shakerSort	77.53	67.17	50.12	100.00	85.35	12.95
shellSort	100.00	0.00	49.69	0.00	104.48	6.97
toUpper	95.10	77.44	50.00	87.20	73.98	32.85

mergeable using CFM, as they meet the structural and functional requirements for merging. However, the third condition cannot be merged due to the recursive function call within its body. It is also important to note that these three conditions are inherently linked. Specifically, the third condition is influenced by the first two: if at least one of the first two conditions evaluates to `true`, the third condition will also evaluate to `true`; conversely, if both of the first two conditions evaluate to `false`, so will the third condition.

By merging the first two conditions, CFM eliminates them, but this can inadvertently affect the predictability of the third condition. Historically, the third condition would benefit from entries in the Pattern History Table (PHT) that were created by the first two conditions, making it highly predictable. When these conditions are removed, the PHT no longer provides the same predictive accuracy for the third condition. This likely explains why the branch misses for the third condition do not decrease as expected and, in fact, slightly increase in the merged code.

In the case of prim, we found that the transformation works at the LLVM IR level, with the branches being removed; however, at the assembly level, the code is modified back to

```
// If left child is larger than root
if (l < n && arr[l] > arr[largest]) {
    largest = l;
}

// If right child is larger than largest so far
if (r < n && arr[r] > arr[largest]) {
    largest = r;
}

// If largest is not root
if (largest != i) {
    int temp = arr[i];
    arr[i] = arr[largest];
    arr[largest] = temp;

    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
}
```

Listing 2: Key control-flow conditions in `heap-sort`.

partially branching code, which causes there to be no significant reduction in misses. This combined with the increase in dynamic instructions executed (since code in the merged part is always run) causes the merged version of prim's algorithm to be slower.

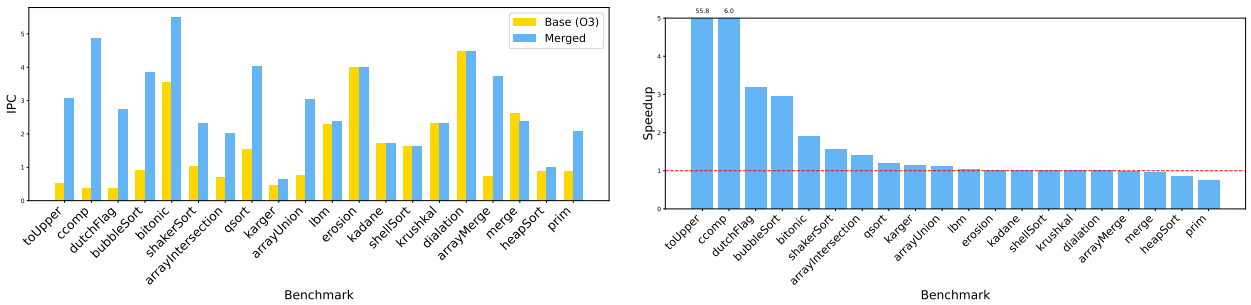
Our experimental results show that the normalized branch misses after applying CFM closely match the theoretical upper bounds, as shown in Table 4.2. In several cases, such as `array-intersect`, `array-merge`, and `bubble-sort`, the experimental miss rates perfectly match or even outperform expectations. However, deviations are observed in benchmarks like `heap-sort` and `prim`.

In `heap-sort`, while CFM successfully merges two of the three key conditions, the third condition, involving a recursive call, remains unmerged. Merging the first two conditions unintentionally disrupts the predictability of the third, leading to a slight increase in branch misses due to weakened Pattern History Table (PHT) entries.

In the case of `prim`, although branches are removed at the LLVM IR level, the compiled assembly reintroduces partial branching, nullifying expected gains. Additionally, the always-executed merged code increases dynamic instruction count, resulting in a slowdown compared to the original version.

Table 4.1 illustrates the Branches Per Kilo Instructions (BPKI) for the branching and merged versions of the benchmarks. A clear reduction in BPKI can be observed across most benchmarks after applying the CFM transformation. Benchmarks like `ccomp`, `array-union`, and `array-intersect` show a significant decrease, highlighting the effectiveness of CFM in removing redundant branches. Notably, benchmarks such as `dilation`, `erosion`, and `kadane` have minimal changes in BPKI due to their inherent control flow being highly optimized even before the transformation. Conversely, `kruskal`, `merge-sort`, and `prim` demonstrate moderate to no reductions, primarily due to the reasons discussed above. It is also worth noting here that for most programs the Misses Per Kilo Instruction (MPKI) is either 0 or close to 0, except in cases of unmergable control flow such as `merge sort` or anomalous cases like `heap sort` and `prim`.

4.1.4 Benchamark Performance



(a) Increase in IPC

(b) Speedup

Figure 4.2: IPC increase and Speedup due to the transformation

Figure 4.3b presents the performance results of our benchmark suite, revealing a geometric mean speedup of 1.57× across all benchmarks. The speedup varies significantly, with improvements as high as 53X in the best case and a performance degradation of 23% in the worst case (prim).

The most notable gain is observed in the `toUpper` benchmark, where performance increases by over 5×. This substantial improvement stems from the removal of a single branch, which enables the compiler to apply more aggressive vectorization optimizations. A similar effect, though less dramatic, is observed in the `bitonic` benchmark. In both cases, the reduced number of executed instructions in the transformed, branchless version confirms that vectorization is more effective. The ability to process more data per instruction, due to simplified control flow, contributes directly to the enhanced performance.

The benchmarks `arrayIntersection`, `arrayMerge`, and `arrayUnion` share similar control-flow and computational characteristics, as they are all based on comparing and merging sorted arrays. Despite these similarities, the benchmarks display notable differences in their respective *instruction ratios* (normalized instructions executed in the transformed versus original versions) and their corresponding *speedups*.

- **Array Intersection** achieves a speedup of **1.40×** with a modest normalized instruction ratio of **1.97**. This reflects efficient merging and a minimal overhead introduced by Control Flow Merging (CFM), owing to relatively fewer operations required for intersection logic.

- **Array Union**, while maintaining the same branch accuracy (**85%**) as `arrayIntersection`, shows a slightly higher normalized instruction ratio (**2.82**) and consequently a lower speedup of **1.10×**. This increase in instructions arises from the additional conditional checks needed to handle union-specific logic, where elements from both arrays must be

considered individually.

- **Array Merge** exhibits the highest instruction ratio (**4.05**) among the three benchmarks and, as expected, demonstrates no performance improvement (**1.00× speedup**). This result indicates that the overhead of additional instructions introduced by CFM outweighs the benefits of branch reduction in this particular case.

These observations emphasize the relationship between the normalized instruction ratio and the resultant speedup. As the instruction ratio increases, the benefits of branch reduction diminish, leading to reduced speedups. For programs like `arrayMerge`, where CFM introduces significant overhead, its effectiveness is limited. Conversely, benchmarks such as `arrayIntersection` demonstrate the potential of CFM to reduce branches without incurring excessive instruction overhead, resulting in measurable performance gains.

4.1.5 Evaluation of CFM for different operations

To evaluate the effectiveness of CFM in mitigating the performance penalties caused by branch mispredictions, we designed an experiment that introduces varying levels of branch accuracy and measures the impact of CFM under these conditions. The experiment focuses on data-dependent branches where the condition relies on the comparison of two arrays. This setup allows us to simulate realistic scenarios while maintaining control over branch accuracy.

Branch Condition and Accuracy Control

To create a data-dependent branch, two large arrays (`test_arr1` and `test_arr2`) are initialized as detailed below. The branch condition compares elements of these arrays. To control

the branch accuracy, we vary the percentage of elements in the arrays that are filled with random values. Specifically:

- **Randomized Section**: A fraction of the arrays, determined by the `RANDOMNESS` parameter, is filled with random values. This introduces unpredictability in the branch condition, reducing branch accuracy.
- **Deterministic Section**: The remaining elements are filled with sequential values that ensure the branch condition is consistently "not taken." This approach introduces a worst-case scenario for a one-sided branch, providing a conservative estimate of the potential speedup from CFM.

```
void setup(int *test_arr1, int *test_arr2) {
    int random_size = (int)(SIZE * (float)RANDOMNESS / 100);
    for (int i = 0; i < random_size + 1; i++) {
       test_arr1[i] = rand() % 2;
       test_arr2[i] = rand();
    }
    for (int i = random_size + 1; i < SIZE + 1; i++) {
       test_arr1[i] = i;
       test_arr2[i] = i + 1;
    }
}
```

Listing 3: Setup function to initialize arrays for branch accuracy experiments.

This benchmark is compiled using the `-O2` optimization level to evaluate the effect of CFM. Both the original (branching) and transformed (branchless) versions are executed to assess the resulting performance gains.

The experiments are divided into three major cases to capture the impact of CFM on different types of instructions: arithmetic logic unit (ALU) operations and memory operations. These two groups are particularly significant because they represent the majority of compu-

tational workloads in real-world programs. Of this memory operations are further divided into operations that are likely to be cache hits, vs operations likely to be cache misses.

- **Arithmetic Operations (ALU)**: The conditionally executed branch contains a simple multiplication operation as seen in Listing 4. This scenario evaluates how CFM handles data-dependent arithmetic instructions.
- **Memory Operations (Linear Access)**: The branch condition involves linear memory access patterns as seen in Listing 5. This setup tests the impact of CFM on predictable, cache-friendly loads and stores.
- **Memory Operations (Random Access)**: The branch condition involves memory access to random locations as seen in Listing 6. Since random accesses are more likely to result in cache misses, this case evaluates how CFM performs under memory-intensive workloads.

```
#define APPLY_OP_1(x) x = x * 89;
for (int i = 1; i < SIZE; i++) {
    if (arr1[i] > arr2[i])
        APPLY_OP_1(arr1[i]);
}
```

Listing 4: Hard-to-predict branch performing a multiplication operation.

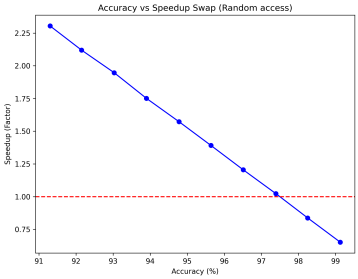
```
#define SWAP_OP_1(x, y) \
    {int temp = x; x = y; y = temp;}
for (int i = 1; i < SIZE; i++) {
    if (arr1[i] > arr2[i])
        SWAP_OP_1(arr1[i], arr2[i]);
}
```

Listing 5: Hard-to-predict branch swapping preloaded array locations.

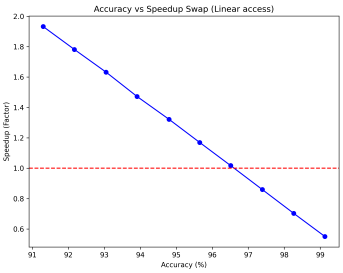
```
#define SWAP_OP_1(x, y) \
    {int temp = x; x = y; y = temp;}
for (int i = 1; i < SIZE; i++) {
    if (arr1[i] > arr2[i])
        SWAP_OP_1(arr1[arr2[i] % SIZE], arr2[i]);
}
```

Listing 6: Hard-to-predict branch swapping a random array location with a preloaded one.

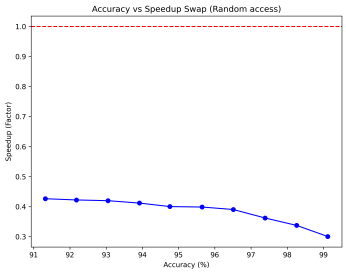
Figure 4.3 highlights how CFM's effectiveness varies with branch accuracy across different operation types. Arithmetic operations, such as multiplication, consistently yield noticeable performance gains, even at high prediction accuracies. Similarly, memory accesses with linear patterns remain profitable up to 96.92% accuracy, though the benefits are slightly tempered due to the inherent cost of memory operations compared to ALU instructions. On the other hand, memory instructions with unpredictable access patterns exhibit degraded performance under CFM. In these cases, the branch removal does not offset the penalties from increased cache misses, leading to overall slowdowns. These findings suggest that CFM is well-suited for compute-intensive tasks and workloads with structured memory behavior, but less beneficial for irregular memory access patterns. Notably, for arithmetic and linearly accessed memory instructions, runtime remains largely unaffected by changes in branch accuracy post-transformation. In contrast, for random memory access, performance deteriorates as branch prediction worsens, likely driven by growing cache miss overheads or improper merging.



(a) Speedup vs. branch accuracy for multiplication



(b) Speedup vs. branch accuracy for array swap (Linear memory access)



(c) Speedup vs. branch accuracy for array swap (Random memory access)

Figure 4.3: Relationship between speedup and branch accuracy for different operations

4.2 Case Studies

4.2.1 X264

X264 is anopen-source video encoding software that implements the H.264/MPEG-4 AVC standard. It is widely used in video streaming and compression applications and is also a part of the SPEC2017 benchmark suite.

Table 4.2: Branch statistics for key lines in `x264`. All values are percentages of the program total.

File	Line #	BrMiss%	CondBrMiss%	CondBr%	TotalBr%	Instr%
`quant.c`	57	47.52	51.32	5.57	7.01	5.26
`quant.c`	49	9.70	9.92	0.77	1.00	0.70
`quant.c`	56	6.67	6.77	5.60	4.87	1.18
`mc.c`	241	2.88	2.53	0.76	0.66	0.15
`me.c`	222	1.40	1.50	0.46	0.38	0.08

Table 4.2 shows the first five locations within the code with the highest number of mispredictions. These five lines alone account for 75% of all mispredictions in the program. Our next step is to analyze these specific code regions in detail to pinpoint the underlying causes of their branch mispredictions, and identify whether they can be removed.

Lines 57,49 from quant.c are reference a macro named QUANT_ONE. This implies that the QUANT_ONE macro shown in Figure 4.4 is a significant source of inefficiency in X264, which is responsible for approximately 65% of total branch misses during execution. This macro checks whether a given value is positive or negative and modifies the coefficient as per the condition. In LLVM IR this converts to a diamond shaped branch, with no control flow edges between the two basic blocks, and is hence fully mergable.

```
QUANT_ONE( coef, mf, f )
{
    if( (coef) > 0 )
        (coef) = (f + (coef)) * (mf) >> 16;
    else
        (coef) = - ((f - (coef)) * (mf) >> 16);
    nz |= (coef);
}
```

Figure 4.4: QUANT_ONE macro.

Further examination reveals that line 56 in quant.c involves a for loop, where the loop condition is likely responsible for frequent branch mispredictions. Additionally, line 241 in mc.c and line 222 in me.c include function calls embedded in if conditions. These constructs are currently beyond the merging capabilities of CFM, which restricts its applicability. Nevertheless, the two remaining locations—lines 57 and 49 in quant.c, associated with the QUANT_ONE macro, are suitable for merging and together contribute to approximately 65% of all branch mispredictions within the program.

As established earlier, CFM is most beneficial when applied to branches that not only suffer from frequent mispredictions but also avoid unpredictable memory access. In this case, both conditions are satisfied: the misprediction rate for the eligible branches is substantial, and the macro in question operates within a loop over a linearly accessed array, as shown in Figure 4.5. This structured access pattern ensures memory predictability, further validating the suitability of CFM for these code regions.

```
static int quant_8x8( int16_t dct[64],
uint16_t mf[64], uint16_t bias[64] )
{
    int nz = 0;
    for( int i = 0; i < 64; i++ )
        QUANT_ONE( dct[i], mf[i], bias[i] );
    return !!nz;
}
```

Figure 4.5: QUANT_ONE Call.

Based on our earlier findings, this macro aligns well with the conditions that favor profitable application of CFM. Specifically, it experiences a high rate of branch mispredictions and operates over a predictable, linear memory access pattern—two factors identified as ideal for successful transformation. When CFM was applied to the corresponding branches in X264, the branch misprediction rate dropped substantially from 3.22% to 1.18%. The Misprediction rate is reduced by 63%, which aligns with the theoretical expection from the previous paragraph.

This significant decrease in mispredictions led to measurable performance gains, resulting in a 12% overall speedup for the application. This case study highlights the effectiveness of CFM in optimizing real-world software, underscoring its practical value in improving execution efficiency by reducing costly branch mispredictions.

4.2.2 XZ

Table 4.3: Branch statistics for key lines in `xz`. All values are percentages of the program total.

File	Line #	BrMiss%	CondBrMiss%	CondBr%	TotalBr%	Instr%
lz_encoder_mf.c	488	14.42	15.23	8.32	7.86	4.64
lz_encoder_mf.c	-1	10.88	11.61	17.44	16.14	15.21
lz_encoder_mf.c	505	12.46	12.93	4.24	3.92	2.23
lz_encoder_mf.c	486	10.58	11.63	4.50	4.08	1.84
lzma_encoder_optimum_normal.c	752	9.90	10.32	2.28	3.66	1.50

The XZ project is a general-purpose data compression tool that implements the LZMA2 algorithm, commonly used for producing .xz archives. Its encoder is designed for high compression ratios and low decompression overhead, making it popular in system-level packaging and distribution pipelines. It is also part of the SPEC CPU 2017 benchmark suite.

Table 4.3 shows the top 5 program locations in XZ ranked by conditional branch mispredictions. These lines span a small number of core source files but together account for approximately 62% of all branch mispredictions observed during profiling. This concentration highlights the narrow set of execution paths responsible for most branch-related stalls and motivates targeted transformations like CFM to improve predictability in these hotspots.

To evaluate the potential for applying Control Flow Merging (CFM), we manually analyzed the top five branch-heavy locations from Table 4.3. Each entry is assessed for whether the surrounding control structure is suitable for safe and effective merging.

- **Line 488 (lz_encoder_mf.c):** This line contains an `if` statement used to return early from a `while` loop. Since the branch causes an early exit and the control flow does not reconverge, it is *not mergable*.
- **Line -1 (lz_encoder_mf.c):** This is a synthetic entry that cannot be precisely mapped to a specific source line by the `addr2line` tool, despite the binary being compiled with `-g`. Such entries typically arise from inlined or loop control branches.
- **Line 505 (lz_encoder_mf.c):** A well-structured `if-else` branch with no early exits or side effects. The control paths reconverge cleanly, making this branch *fully mergable* by CFM.
- **Line 486 (lz_encoder_mf.c):** This line includes a `while` loop inside an `if` condition. The current CFM implementation does not support merging conditionals that

contain loops. Moreover, fully merging a loop would be unprofitable due to the duplication of loop body code. This site is therefore *not mergable*.

- **Line 752 (lzma_encoder_optimum_normal.c):** Similar to line 486, this line contains a `for` loop nested within a conditional. As with `while` loops, such constructs are outside the current scope of CFM. The outer conditional may be simplifiable, but full merging is unsupported and likely unprofitable. This case is *not mergable*.

```
if (pb[len] < cur[len]) {
    *ptr1 = cur_match;
    ptr1 = pair + 1;
    cur_match = *ptr1;
    len1 = len;
} else {
    *ptr0 = cur_match;
    ptr0 = pair;
    cur_match = *ptr0;
    len0 = len;
}
```

Figure 4.6: Mergable branch in line 505 of `lz_encoder_mf.c`

Table 4.4: Performance impact of CFM on `xz`.

Configuration	Time (s)	Branch Misprediction %	TMA Mem Bound %	IPC
Base (`clang -O3`)	397.99	3.58%	16.5%	2.00
CFM (complete merging)	482.16	3.04%	27.2%	1.94
CFM (targeted merging)	473.40	3.29%	27.3%	1.81

Table 4.4 summarizes the runtime results of applying CFM on XZ. We can see that both targetted and indiscriminate application of CFM cause a significant performance penalty, of around 20%, despite a modest reduction in branch misprediction of around 7% and 14% respectively. This can be directly attributed to the tma (Top-Down Microarchitecture Analysis) memory bound statistic provided by perf, which provides an insight into overhead

due to memory operations, with both of the merged versions having a significantly higher number of stalls due to memory access.

This serves as a reminder of the most important challenge with CFM, it is hard to statically analyze when CFM could hinder loads, especially in cases where the memory location gets selected. Selecting the memory location can hinder prefetching performed by the CPU and cause a significant slowdown.

4.2.3 MCF

The `mcf` benchmark is part of the SPEC CPU 2017 suite and represents a combinatorial optimization workload focused on solving minimum-cost flow problems in networks. It is computation-heavy and characterized by sparse, pointer-based data structures and frequent conditional checks over dynamic control paths. These properties make `mcf` a valuable candidate for evaluating branch reduction techniques, as its control flow contains a mix of loop constructs and branching decisions that are difficult to predict and often poorly optimized by standard compiler passes.

Table 4.5: Branch statistics for key lines in `mcf`. All values are percentages of the program total.

File	Line #	BrMiss%	CondBrMiss%	CondBr%	TotalBr%	Instr%
`pbeampp.c`	68	55.39	53.42	15.17	12.36	13.23
`pbeampp.c`	53	13.73	15.03	9.60	7.54	3.28
`pbeampp.c`	70	11.96	12.48	8.07	6.36	1.39
`pbeampp.c`	54	4.77	5.31	7.99	6.10	2.81
`implicit.c`	358	4.25	4.23	2.08	1.63	1.39

Table 4.5 highlights the top five code locations in `mcf` with the highest share of conditional branch mispredictions. Remarkably, these five lines alone account for nearly 90% of all conditional branch mispredicts in the program, despite being spread across only two source

files. This stark concentration underscores how a small set of control-flow decisions dominates the branch behavior of the application, making them highly attractive targets for transformations like CFM.

To assess the applicability of Control Flow Merging in these scenarios, we performed a manual analysis of the control structures surrounding each of the top five lines. This helps determine whether each instance is amenable to transformation via CFM and what structural features—such as loops, early exits, or side effects—might prevent merging.

The branch at line 68 in `pbeampp.c` is a simple conditional `return`, responsible for over half of all conditional mispredictions in `mcf`. Despite its simplicity, this early exit breaks control flow reconvergence, making it currently *unmergable* by CFM. Line 70 is also a conditional return and is similarly hindered. In contrast, other hotspots, such as line 53/54, involve compound logical conditions (`&&`, `||`) within return statements. These are structurally more complex but potentially *partially mergable*, depending on whether the control flow converges after evaluation. These two patterns—early exits and logic-heavy returns—capture the main challenges for CFM in `mcf`. Line 358 of MCF (Image 4.7) is however fully mergable, and responsible for 4.25% of the misprediction in the program.

```
if (newarc[cmp - 1].flow < newarc[cmp].flow)
    cmp++;
```

Figure 4.7: A conditional update in `mcf` that is fully mergable via CFM. The control flow is linear and side-effect free.

Table 4.6: Performance Metrics Comparison Across `mcf` Executables

Executable	**Runtime (s)**	**IPC**	**Branch Mispred. Rate (%)**	**Backend Bound (%)**
`Baseline`	2744.52	1.60	1.58	28.8
`CFM Targeted`	2756.43	1.59	1.58	26.9
`CFM Aggressive`	3951.07	1.96	1.34	38.6
`CFM only in loops`	3554.98	2.05	1.36	34.4

Table 4.6 presents the impact of applying CFM to the `mcf` benchmark under various config-

urations. Notably, all CFM-enhanced versions perform worse than the baseline in terms of runtime, despite achieving higher IPC. Both "CFM Aggressive" and "CFM only in loops" configurations exhibit significant slowdowns, even though their IPCs improve to 1.96 and 2.05, respectively. This can likely be attributed to the marked increase in backend-bound stalls, especially in the aggressive case, which reaches 38.6%. This suggests that the additional dynamic instruction overhead from executing both paths in merged regions exacerbates memory or ALU contention.

The "CFM Targeted" variant presents a particularly revealing scenario. Here, specific regions known to contribute to performance degradation were merged, including a branch responsible for approximately 4% of all mispredictions and the `spec_qsort` function. Despite successful IR-level merges in these functions, the overall branch misprediction rate remains unchanged at 1.58%. This strongly suggests that subsequent backend compiler passes may be reintroducing conditional control flow or otherwise undoing CFM's effects, either fully or partially. This outcome highlights the importance of preserving merged structures through backend code generation and supports the need for further investigation into backend interactions with IR-level transformations like CFM.

4.2.4 Oggenc

The `oggenc` benchmark, sourced from the LLVM test suite, is an audio encoding application that implements the Ogg Vorbis compression format. It is representative of real-world multimedia workloads and involves a mix of signal processing, buffer manipulation, and control-heavy logic. `oggenc` operates on audio data streams, making it valuable for evaluating compiler transformations in practical settings. Its control flow is shaped by a combination of format-specific encoding steps and dynamic runtime conditions, resulting in

numerous branches, many of which are input-dependent and prone to misprediction. This makes `oggenc` a compelling case study for evaluating the effectiveness of Control Flow Merging (CFM) in reducing branch-related performance penalties.

Table 4.7: Branch statistics for key lines in `oggenc`. All values are percentages of the program total.

File	Line #	BrMiss	CondBrMiss	CondBr	TotalBr	Instr
`oggenc.c`	54884	8.53	9.59	1.81	1.23	0.24
`oggenc.c`	48752	10.08	8.22	3.85	3.04	1.19
`oggenc.c`	48876	1.55	5.48	3.02	2.78	0.90
`oggenc.c`	54894	3.88	5.48	2.11	1.49	0.59
`oggenc.c`	48809	11.63	5.48	1.36	1.49	0.48
`oggenc.c`	54725	0.78	5.48	1.36	0.84	0.45
`oggenc.c`	49288	3.10	5.48	0.30	0.65	0.05
`oggenc.c`	48755	4.65	4.11	8.60	6.27	2.03
`oggenc.c`	55728	7.75	4.11	1.66	1.68	0.65
`oggenc.c`	55687	2.33	4.11	0.68	0.84	0.27

Table 4.7 shows the top 10 code locations in `oggenc.c`, the single-source benchmark distribution of `oggenc` provided as part of the LLVM test suite. These lines together account for approximately 55% of all conditional branch mispredictions, a relatively modest concentration compared to other benchmarks analyzed in this work. The mispredicted branches in `oggenc` are more widely distributed across the codebase, reflecting the program's complex and scattered control flow. Despite the dispersion, many of these branches are structurally simple or localized, indicating strong potential for applying Control Flow Merging (CFM). A targeted application of CFM in these hotspots may still yield improvements in predictability and instruction throughput.

We manually analyzed the control structures corresponding to the most misprediction prone lines in `oggenc.c` to evaluate their suitability for transformation via Control Flow Merging (CFM). Lines 54884 and 54725 represent straightforward conditional assignments and are

fully mergable, likely even by existing LLVM passes such as SimplifyCFG. Other lines, such as 48752, involve one-sided branches—patterns where CFM offers a clear advantage, as such structures are typically left unoptimized by existing passes. Lines like 48876 and 49288 feature early `return` statements guarded by loop conditions, which complicate merging due to disrupted control flow. Similarly, lines 48809 and 48755 involve `break` statements or function-level exits, making them unmergable under current CFM constraints. Finally, the branches at lines occur contain nested loops, meaning that meging even if done would be unprofitable. This diversity in branching behavior demonstrates both the limitations of existing passes and the nuanced opportunities available to CFM in realistic, control-heavy codebases like `oggenc`.

Table 4.8: Performance Metrics Comparison Across Executables

Executable	**Runtime (s)**	**IPC**	**Branch Mispred. Rate (%)**	**Backend Bound (%)**
`-O3 (baseline)`	58.49	2.52	3.74	9.7
`CFM Targetted`	55.61	2.65	3.51	9.3
`Force CFM`	58.44	2.78	3.39	11.2
`CFM only in loops`	57.51	2.79	3.43	10.5

Table 4.8 highlights the impact of Control Flow Merging (CFM) under different application strategies. All CFM-transformed variants exhibit higher Instructions Per Cycle (IPC) than the baseline, indicating improved pipeline utilization. IPC increases with more widespread application of CFM, with "Force CFM" and "CFM only in loops" both achieving similar IPC improvements. Likewise, branch misprediction rates show a consistent decline as more control-flow regions are merged, demonstrating CFM's effectiveness in eliminating difficult-to-predict branches.

Despite these microarchitectural improvements, the runtime benefits do not scale proportionally. For instance, although both "Force CFM" and "CFM only in loops" yield higher IPC and lower misprediction rates, their runtimes remain comparable to the baseline. This suggests that excessive application of CFM may introduce backend pressure—such as in-

creased memory traffic or ALU utilization—that offsets the gains from improved control flow.

In contrast, the "CFM Targeted" configuration achieves a 4.9% speedup over the baseline, with only a modest 6% reduction in misprediction rate. This selective strategy focuses solely on the functions `inspect_error` and `seed_curve`, which contain the most costly branches at lines 48752 and 54884, respectively. The substantial performance benefit from targeting these specific regions reinforces the presence of disproportionately expensive "bad branches," whose elimination yields outsized runtime improvements. These findings underscore that CFM is most effective when applied judiciously to performance-critical control paths rather than uniformly across the codebase.

Chapter 5

Conclusion and Future work

5.0.1 Conclusion

In this work, we introduced *Control Flow Merging (CFM)*, a novel compiler transformation aimed at reducing branch misprediction penalties by converting divergent control-flow paths into data flow using `select` instructions. Unlike existing LLVM optimization passes such as `InstCombine`, `SimplifyCFG`, and function merging, CFM is capable of operating on regions that contain side-effecting instructions. This allows it to handle a broader class of code, including complex constructs like *one-sided branches*, which can contribute misprediction penalties comparable to conventional two-sided branches.

Our results demonstrate that CFM is particularly effective on smaller, branch-sensitive kernels, where even modest rates of misprediction can lead to significant performance degradation. By eliminating control-flow divergence in these hot regions, CFM offers meaningful improvements in predictability and runtime behavior that are not achievable with existing LLVM passes.

However, CFM is not a universally beneficial transformation. When applied to regions that suffer from poor memory locality or frequent cache misses, the additional instruction overhead from merging can result in performance slowdowns. This highlights the need for more selective and informed application of the technique.

5.0.2 Future Work

Partial Merging of Basic Blocks

While Control Flow Merging (CFM) is effective at eliminating branches and reducing misprediction penalties, applying it indiscriminately to entire basic blocks is not always profitable. In particular, merging very large blocks can introduce significant instruction overhead. Further, certain control flow patterns or instruction types—such as function calls, loops and early returns cannot be merged by the transformation.

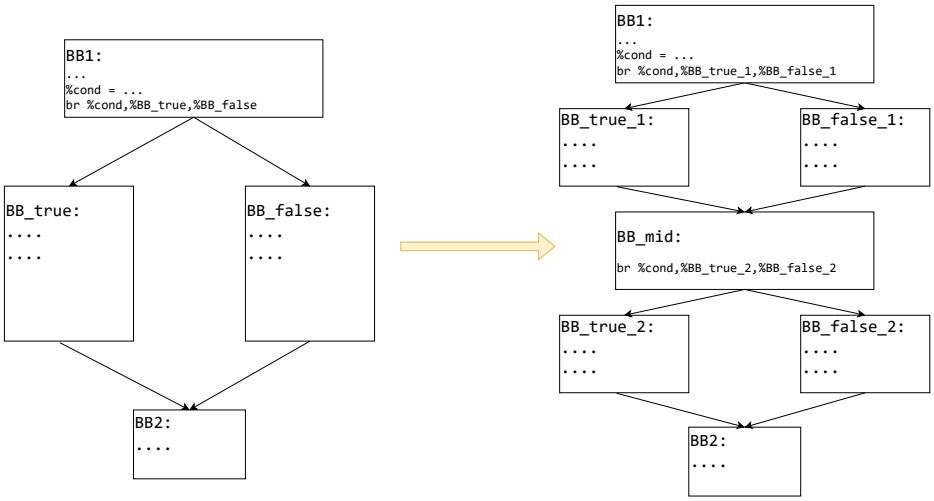


Figure 5.1: control flow restructuring - Block Split

To address these challenges, we introduced a strategy called *partial merging*, illustrated in Figures 5.1 and 5.2. Instead of attempting to merge entire blocks, we first split them into two parts at a suitable boundary: a merge-friendly prefix and a suffix that may contain unhandled or unprofitable constructs. This is shown in Figure 5.1, where the blocks BB1 and BB2 are divided into two basic blocks, respectively, with BB_mid inserted between them to maintain program semantics.

After this split, CFM is applied to the prefix blocks, as shown in Figure 5.2. The result is

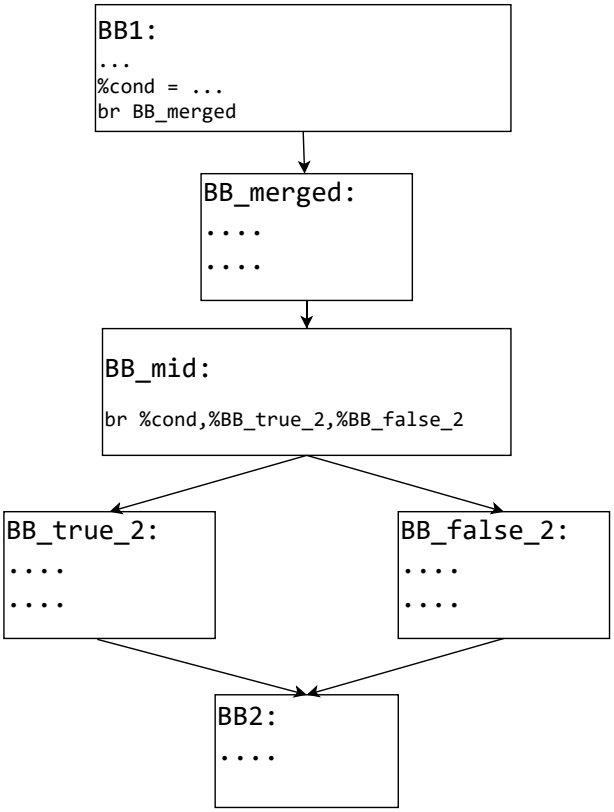


Figure 5.2: Partial merge of control paths

a merged block that evaluates the original branch condition only once (in BB1), and propagates this information forward to select the appropriate continuation. Importantly, the new branch introduced after the merged prefix (BB_merged) does not suffer from mispredictions, because the branch direction has already been resolved unconditionally. This approach effectively reduces control divergence and misprediction costs even in cases where full merging is infeasible.

Partial merging, therefore extends the applicability of CFM to more complex code scenarios, making it a useful addition to CFM.While implementing the partial split is a straightforward process, which has been successfully implemented as an extension to the transformation, reasoning about its performance implications is considerably more complex. If the merged portion of the block is too small, the branch condition may not be evaluated in time for use

in the second half (e.g., `BB_mid`), leading to renewed mispredictions. On the other hand, if the merged region is too large, it may introduce unnecessary duplication or register pressure, degrading performance.

To approximate an optimal split point, a simple cost model was developed that relied on estimated instruction latencies derived from the compiler's middle-end heuristics. However, this model often proved insufficient. In practice, it often produced suboptimal splits, either resulting in mispredictions at `BB_mid` or performance regressions due to merging overly large block prefixes. Future work could improve upon this by designing a more accurate cost model that takes into account instruction latencies, data dependencies, and even dynamic execution profiles to guide partial merging more effectively.

Loop Peeling to enable merging

Loop peeling is a transformation that extracts a fixed number of iterations from the beginning of a loop and hoists them outside the loop body. This is typically done by the compiler to simplify loop conditions, eliminate edge cases, or enable more aggressive optimizations such as vectorization or unrolling on the remaining loop.

Figure 5.3 shows an example of loop peeling. In the original loop (Figure 5.3a), the decrement operation on `a[i]` occurs repeatedly while the condition `a[i] > 0` holds. In the transformed version (Figure 5.3b), the compiler has peeled one iteration by adding a conditional execution of the body before the loop begins. This allows the compiler to handle the first iteration differently or with fewer control-flow checks inside the main loop.

In the case of a simple `while` loop with a monotonically decreasing or predictable condition (e.g., decrementing a counter to zero), modern branch predictors quickly learn the pattern and the number of mispredictions tends to be constant—typically one or two—depending

```
while (x > 0){
    x--;
}
```

(a) Original loop

```
if (x > 0) x--;
while (x > 0){
    x--;
}
```

(b) After peeling one iteration

Figure 5.3: Loop peeling transformation

on the initial state of the predictor.

However, when such a loop is nested within an outer loop (Figure 5.4), the branch predictor must re-evaluate the inner loop's exit condition every time the outer loop restarts. Most history-based predictors (even sophisticated ones like TAGE) struggle to predict the final iteration of a loop, where the branch switches from taken to not-taken, causing a predictable misprediction at each loop exit, especially in nested loops where the inner loop is restarted many times. Since the predictor cannot infer loop trip counts dynamically, this results in a linear number of mispredictions in the number of outer loop iterations.

```
for (int i = 1; i < n; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
```

Figure 5.4: Nested loop in Insertion Sort. Mispredictions occur once per outer iteration due to loop exit behavior.

Here, the inner `while` loop is executed once per element of `a` and the total number of mispredictions grows as $\mathcal{O}(N)$ in contrast to the constant misprediction cost of a standalone loop. This behaviour is seen in some larger workloads, where loops can contribute significantly to mispredictions.

While the loop structure in 5.3a, cannot be merged due to the backedge, once peeled, the

first few iterations of the loop can be merged, and this might be quite useful in reducing mispredictions, especiaally in cases where the initial iterations of the loop are mispredicted.

Software Pre-fetching

Software prefetching allows developers to explicitly insert prefetch instructions (e.g., `__builtin_prefetch` in GCC/Clang) to bring data into the cache before it is accessed. This is particularly useful when compiler transformations like Control Flow Merging (CFM) restructure control flow in a way that obscures predictable memory access patterns. In such cases, hardware prefetchers may fail to detect regular strides or may be delayed in issuing prefetches. By manually prefetching data that will be accessed in merged branches software prefetching can reduce memory access latency and mitigate performance regressions introduced by CFM. This makes it a valuable complementary technique, especially in memory-bound loops with merged control flow. Importantly, since prefetching is treated as a hint, issuing a prefetch to an invalid memory address does not raise an exception and is silently ignored, preserving program safety.

The code segment in Figure 5.5 corresponds to a performance-critical section within the XZ utility, where poor branch behavior was previously identified. The critical branch occurs at line 30 in the listing. This conditional controls the update of internal pointers to either pair or pair + 1, both of which involve memory dereferencing.

This region is known to be memory-bound, which previously caused significant performance degradation when Control Flow Merging (CFM) was applied. To address this, we inserted explicit prefetch instructions (___builtin_prefetch) immediately after computing the pair address. Since pair and pair + 1 are both accessed within the conditional, prefetching them ensures that data is likely to reside in cache by the time it's needed.

This change improves performance notably: execution time drops from 473 seconds to 411 seconds when CFM is enabled. However, the unmerged version, without any merging transformation, sees a slight performance regression, increasing from 397 to 415 seconds. This suggests that while prefetching is generally beneficial, in the unmerged case it might introduce suboptimal cache behavior or unnecessary memory pressure.

Crucially, with prefetching, the merged version no longer suffers from the drastic performance penalty it did earlier. This demonstrates that prefetching can effectively resolve CFM-induced slowdowns in memory-bound contexts, validating the transformation's safety and benefit in such cases.

Profile Guided Optimizations

An important direction for future work is the integration of Control Flow Merging (CFM) into a *Profile-Guided Optimization (PGO)* framework. While CFM can effectively reduce branch mispredictions, its impact on memory behavior and instruction-level parallelism is highly context-dependent. Relying solely on static analysis to decide when to apply CFM is insufficient, particularly in memory-bound workloads where cache effects dominate performance.

By leveraging dynamic profiling data, a PGO-enhanced implementation of CFM can make more informed, data-driven decisions. Profiling can reveal real-world branch predictability, memory access patterns, and execution frequency information that enables the compiler to assess the profitability of merging control paths on a case-by-case basis. Embedding CFM into LLVM's existing PGO infrastructure would allow it to be applied selectively and opportunistically, ensuring that it improves performance only in scenarios where the benefits outweigh the costs.

Extension to generalized Single Entry Single Exit

Another important extension is to enable CFM to operate on *single-entry single-exit (SESE)* regions. This generalization would broaden the scope of the transformation beyond simple if-else constructs, making it applicable to larger and more complex control-flow subgraphs.

However, this generality also introduces new performance considerations. The effectiveness of CFM hinges on minimizing code duplication, and as the complexity of the merged region increases, so does the potential for duplicating instructions across divergent paths. This can lead to redundant execution, increased register pressure, and instruction cache stress. Therefore, when targeting SESE regions, it becomes doubly important to consider profitability checks and incorporate cost models that account for the structural complexity of the region. Without such safeguards, the broader applicability of CFM could come at the expense of degraded performance.

```
uint32_t *const pair = son + ((cyclic_pos - delta
        + (delta > cyclic_pos ? cyclic_size : 0))
        << 1);

__builtin_prefetch(pair, 0, 0);
__builtin_prefetch(pair + 1, 0, 0);

const uint8_t *const pb = cur - delta;
uint32_t len = my_min(len0, len1);

if (pb[len] == cur[len]) {
    while (++len != len_limit)
        if (pb[len] != cur[len])
            break;

    if (len_best < len) {
        len_best = len;
        matches->len = len;
        matches->dist = delta - 1;
        ++matches;

        if (len == len_limit) {
            *ptr1 = pair[0];
            *ptr0 = pair[1];
            return matches;
        }
    }
}

if (pb[len] < cur[len]) {
    *ptr1 = cur_match;
    ptr1 = pair + 1;
    cur_match = *ptr1;
    len1 = len;
} else {
    *ptr0 = cur_match;
    ptr0 = pair;
    cur_match = *ptr0;
    len0 = len;
}
```

Figure 5.5: Branch traversal in binary tree matcher with prefetching optimization

Bibliography

[1] Muawya Al-Otoom, Elliott Forbes, and Eric Rotenberg. Exact: explicit dynamic-branch prediction with active updates. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, page 165–176, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300445. doi: 10.1145/1787275.1787321. URL https://doi.org/10.1145/1787275.1787321.

[2] Jayvant Anantpur and R. Govindarajan. Taming control divergence in gpus through control flow linearization. In *Proceedings of the 2014 International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2014. doi: 10.1145/2544137.2544146.

[3] Paul Biggar, Nicholas Nash, Kevin Williams, and David Gregg. An experimental study of sorting and branch prediction. *ACM J. Exp. Algorithmics*, 12, June 2008. ISSN 1084-6654. doi: 10.1145/1227161.1370599. URL https://doi.org/10.1145/1227161.1370599.

[4] Chien-Ming Chen, Yunn-Yen Chen, and Chung-Ta King. Branch merging for effective exploitation of instruction-level parallelism.

[5] Sana Damani and Vivek Sarkar. Common subexpression convergence: A new code optimization for simt processors. In *Proceedings of the 2014 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 421–432. IEEE, 2014. doi: 10.1145/2628071.2628096.

[6] Aniket Deshmukh, LingzheChester Cai, and Yale N. Patt. Timely, efficient, and accu-

rate branch precomputation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 480–492, 2024. doi: 10.1109/MICRO61859.2024.00043.

[7] Amr Elmasry and Jyrki Katajainen. Branchless search programs. In *Experimental Algorithms, SEA 2013*, volume 7933 of *LNCS*, page 127–138. Springer, 2013.

[8] Stijn Eyerman, James E. Smith, and Lieven Eeckhout. Characterizing the branch misprediction penalty. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–58. IEEE, 2006. URL https://users.elis.ugent.be/~leeckhou/papers/ispass06-eyerman.pdf.

[9] M. U. Farooq, Khubaib, and L. K. John. Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 59–70. IEEE, 2013. doi: 10.1109/HPCA.2013.6522307.

[10] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 74–85. IEEE, 2008. doi: 10.1109/HPCA.2008.4658629.

[11] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A. Jiménez. Bit-level perceptron prediction for indirect branches. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 27–38, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366694. doi: 10.1145/3307650.3322217. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/3307650.3322217.

[12] Maziar Goudarzi, Reza Azimi, Julian Humecki, Faizaan Rehman, Richard Zhang, Chi-

rag Sethi, Tanishq Bomman, and Yuqi Yang. By-software branch prediction in loops. *arXiv preprint arXiv:2305.08317*, 2023. URL https://arxiv.org/abs/2305.08317.

[13] Longkun Guo, Kewen Liao, Hong Shen, and Peng Li. Efficient approximation algorithms for computing k disjoint restricted shortest paths. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 62–64. ACM, 2015. doi: 10.1145/2755573.2755580.

[14] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. Opportunistic early pipeline re-steering for data-dependent branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 305–316, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380751. doi: 10.1145/3410463.3414628. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/3410463.3414628.

[15] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. Opportunistic early pipeline re-steering for data-dependent branches. In *Proceedings of the 29th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 305–316. ACM, 2020. doi: 10.1145/3410463.3414628.

[16] Milad Hashemi, Onur Mutlu, and Yale N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. doi: 10.1109/MICRO.2016.7783764.

[17] Mohammad Hashemi, Onur Mutlu, and Yale N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016. doi: 10.1109/MICRO.2016.7783742.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Burlington, MA, 6th edition, 2017. ISBN 9780128119051. URL https://acs.pub.ro/~cpop/SMPA/Computer%20Architecture%2C%20Sixth%20Edition_%20A%20Quantitative%20Approach%20%28%20PDFDrive%20%29.pdf. See Section 3.3 "Reducing Branch Costs With Advanced Branch Prediction" (p. 182-191).

[19] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 197–206. IEEE, 2001. doi: 10.1109/HPCA.2001.903263.

[20] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems (TOCS)*, 20(4):369–397, 2002. doi: 10.1145/571637.571640.

[21] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A. Jiménez, and Baris Kasikci. Whisper: Profile-guided branch misprediction elimination for data center applications. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '22, page 19–34. IEEE Press, 2023. ISBN 9781665462723. doi: 10.1109/MICRO56248.2022.00017. URL https://doi-org.ezproxy.lib.vt.edu/10.1109/MICRO56248.2022.00017.

[22] William C. Kreahling, David B. Whalley, Mark W. Bailey, Xin Yuan, Gang-Ryung Uh, and Robert van Engelen. Branch elimination via multi-variable condition merging. In *Euro-Par 2003: Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 261–270. Springer, 2003. doi: 10.1007/978-3-540-45209-6_40. URL https://link.springer.com/chapter/10.1007/978-3-540-45209-6_40.

[23] Rakesh Kumar and Boris Grot. Shooting down the server front-end bottleneck. *ACM*

Trans. Comput. Syst., 38(3–4), January 2022. ISSN 0734-2071. doi: 10.1145/3484492. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/3484492.

[24] Geoff Langdale and Daniel Lemire. Parsing gigabytes of json per second. *arXiv preprint arXiv:1902.08318*, 2019.

[25] Daniel Lemire. Benchmarking is hard: processors learn to predict branches, 2019. URL https://lemire.me/blog/2019/10/16/benchmarking-is-hard-processors-learn-to-predict-branches/. Accessed: 2024-11-07.

[26] Chit-Kwan Lin and Stephen J. Tarsa. Branch prediction is not a solved problem: Measurements, opportunities, and future directions, 2019. URL https://arxiv.org/abs/1906.08170.

[27] Pierre Michaud. An alternative tage-like conditional branch predictor. *ACM Trans. Archit. Code Optim.*, 15(3), August 2018. ISSN 1544-3566. doi: 10.1145/3226098. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/3226098.

[28] Simon Moll and Sebastian Hack. Partial control-flow linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 560–574. ACM, 2018. doi: 10.1145/3192366.3192413.

[29] Anand Naithani, Jaume Feliu, Ahmad Adileh, and Lieven Eeckhout. Precise runahead execution. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 397–410. IEEE, 2020. doi: 10.1109/HPCA47549.2020.00040.

[30] Dorit Nuzman, Ayal Zaks, and Ziv Ben-Zion. If-convert as early as you must. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Con-*

struction, CC 2024, page 26–38, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705076. doi: 10.1145/3640537.3641562. URL https://doi.org/10.1145/3640537.3641562.

[31] Stephen Pruett and Yale N. Patt. Branch runahead: An alternative to branch prediction for impossible to predict branches. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 804–815. IEEE, 2021. doi: 10.1145/3466752.3480053. URL https://dl.acm.org/doi/10.1145/3466752.3480053.

[32] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction – extended version, 2022. URL https://arxiv.org/abs/2107.05681.

[33] Charitha Saumya, Rohan Gangaraju, Kirshanthan Sundararajah, and Milind Kulkarni. Targeted control-flow transformations for mitigating path explosion in dynamic symbolic execution, 2023. URL https://arxiv.org/abs/2308.01554.

[34] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148. ACM/IEEE Computer Society, 1981. URL https://courses.cs.washington.edu/courses/cse590g/04sp/Smith-1981-A-Study-of-Branch-Prediction-Strategies.pdf.

[35] Vinesh Srinivasan, Rangeen Basu Roy Chowdhury, and Eric Rotenberg. Slipstream processors revisited: Exploiting branch sets. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2020. doi: 10.1109/ISCA45697.2020.00020.

[36] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the Ninth Inter-*

national Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, page 257–268, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581133170. doi: 10.1145/378993.379247. URL https://doi.org/10.1145/378993.379247.

[37] Rajat Tiwari, Mohammad Shahrad, Abhishek Bhattacharjee, and David Wentzlaff. Load driven branch predictor. *arXiv preprint arXiv:2009.09064*, 2020. URL https://arxiv.org/abs/2009.09064.

[38] Luis A. Q. Villon, Raphael G. Azevedo, Leonardo T. de Souza, Bruno C. Silva, Renato P. Ribas, and Altamiro A. Susin. A conditional branch predictor based on weightless neural networks. *Neurocomputing*, 523:126–136, 2023. doi: 10.1016/j.neucom.2022.11.095.

[39] Siavash Zangeneh, Lizy K. John, and Andreas Gerstlauer. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 118–130. IEEE, 2020. doi: 10.1109/MICRO50266.2020.00020.

[40] Peng Zhou, Soner Önder, and Steve Carr. Fast branch misprediction recovery in out-of-order superscalar processors. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, page 41–50, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931678. doi: 10.1145/1088149.1088156. URL https://doi-org.ezproxy.lib.vt.edu/10.1145/1088149.1088156.