HyBF: A Hybrid Branch Fusion Strategy for Code Size Reduction

Rodrigo C. O. Rocha*†
University of Edinburgh
UK
Rodrigo.Rocha@huawei.com

Charitha Saumya*
Purdue University
USA
cgusthin@purdue.edu

Kirshanthan Sundararajah
Purdue University
USA
ksundar@purdue.edu

Pavlos Petoumenos
University of Manchester
UK
pavlos.petoumenos@manchester.ac.uk

Milind Kulkarni
Purdue University
USA
milind@purdue.edu

Michael F. P. O'Boyle
University of Edinburgh
UK
mob@inf.ed.ac.uk

Abstract

Binary code size is a first-class design consideration in many computing domains and a critical factor in many more, but compiler optimizations targeting code size are few and often limited in functionality. When size reduction opportunities are left unexploited, it results in higher downstream costs such as memory, storage, bandwidth, or programmer time.

We present *HyBF*, a framework to manage code merging techniques that target conditional branches (*i.e., if-then-else*) with similar code regions on both paths. While such code can be easily and profitably merged with little control flow overhead, existing techniques generally fail to fully handle it. Our work is inspired by branch fusion, a technique for merging similar code in *if-then-else* statements, which is aimed at reducing thread divergence in GPUs. We introduce two new branch fusion techniques that can be applied to almost any *if-then-else* statement and can uncover many more code merging opportunities. The two approaches are mostly orthogonal and have different limitations and strengths. We integrate them into a single framework, *HyBF*, which can choose the optimal approach on a per branch basis to maximize the potential of reducing code size.

Our results show that we can achieve significant code savings on top of already heavily optimized binaries using state-of-the-art code size optimizations. Over 61 benchmarks, we reduce the code size on 43 of them. That reduction typically ranges from a few hundred to a few thousand bytes, but for specific benchmarks, it can be substantial and as high as 4.2% or 67 KB.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: Branch Fusion, Function Merging, LLVM, Compiler Optimization, Code-Size Reduction

ACM Reference Format:
Rodrigo C. O. Rocha, Charitha Saumya, Kirshanthan Sundararajah, Pavlos Petoumenos, Milind Kulkarni, and Michael F. P. O'Boyle. 2023. *HyBF*: A Hybrid Branch Fusion Strategy for Code Size Reduction. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction (CC '23), February 25–26, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578360.3580267

1 Introduction

Code size is critical for computing systems with constrained resources. These systems operate under limited addressable memory, storage, and bandwidth from tiny embedded devices up to cloud servers. As programs gain new features over time, continuously growing in size and complexity [4, 20], they can eventually become excessively large relative to the given constraints. This has a detrimental effect on the system, ultimately causing its failure. In such scenarios, compiler optimizations for reducing the application's footprint are essential [2, 4, 18, 32, 33, 41].

Compilers provide optimizations that are tailored for code reduction [13, 29], including dead-code elimination [39], common sub-expression elimination [8], redundancy elimination [3], and constant propagation [42]. Recent developments on code size optimizations have focused on merging equivalent code to avoid duplicates, which include function merging, function outlining, and loop rolling. Function merging identifies functions with sufficient similarity and merges them into a single function [14, 28–30]. Function outlining identifies equivalent basic blocks across all functions and extracts them into a function, replacing all the copies by a

*The first two authors contributed equally to this work.
†Now at Huawei.

CC '23, February 25–26, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0088-0/23/02…$15.00
https://doi.org/10.1145/3578360.3580267

function call [4]. Loop rolling transforms equivalent instructions from a single block into a loop [27].

Despite all these efforts, state-of-the-art techniques still cannot fully exploit code similarity to reduce code size. In this paper, we identify such a missed opportunity: reducing code size by merging similar code in branches that form *if-then-else* constructs. This technique, known as *branch fusion*, was originally proposed as an optimization for reducing thread divergence in GPUs [10, 31]. Our key insight is that branch fusion can be adapted to reducing code size. The main barrier for that is the simplicity of existing techniques. Branch fusion [10] only merges highly similar single-block branch paths, while control-flow melding (DARM) [31] only works on branch paths that contain simple control-flow regions, *e.g.,* nested *if-then*,*if-then-else* or *natural loops*, that are isomorphic. DARM might be sufficient to cover certain important GPU kernels, but its was not designed for code size reduction and, it is not capable of handling complex control-flow regions present in real-world CPU programs.

Our work introduces two novel branch fusion approaches that overcome these limitations. The first one, Control-flow Melding for Code Size (CFM-CS), adapts DARM [31] to reduce code size in CPU programs while extending it to support any complex Single-Entry-Single-Exit (SESE) regions that can be present in real programs, including unstructured control-flow. The second one, SEME-Fusion, generalizes function merging [28], enabling it to also work as a branch fusion technique capable of merging any pair of SEME regions inside a conditional branch using its flexible matching approach allows us to uncover merging opportunities in branches whose divergent paths have little structural similarity. The two techniques have different strengths and limitations, uncovering different code-saving opportunities. We combine them into a unified framework, *HyBF*, that chooses which one to apply on a branch-to-branch basis, maximizing their potential.

The experimental results show that our approach is capable of significant code size reduction: up to 67KB and 3.1KB on average, much higher than any individual branch fusion approach on its own. We demonstrate that significant gains are obtained on top of highly optimized programs including state-of-the-art code size optimizations, such as function merging [29, 36]. The compile-time overhead we pay for this is low, less than 8% slowdown for half of the benchmarks, 15.7% on average. The effect on the performance of the generated binaries is negligible.

Our main contributions are:

- We are the first to employ branch fusion and control-flow melding for code size reduction.
- We propose the first branch fusion technique capable of merging any pair of SEME regions, regardless of their structural similarity.
- We propose *HyBF*, a novel framework that combines multiple branch fusion techniques to leverage their individual strengths in different scenarios.
- An implementation of *HyBF* in LLVM that is publicly available[1].
- We show the effectiveness of our approach on full benchmark programs as well as individual functions extracted from real-world code.

2 Background and Motivation

In this section, we contrast the existing optimizations for code size reduction on conditional branches. We also motivate the need for a new branch fusion strategy tailored for code size optimizations.

2.1 Code Motion

The two important *code motion* optimizations for code reduction are *code hoisting* and *code sinking* [7, 39]. Both techniques replace multiple equivalent expressions (*i.e.,* expression that produce the same value) with a single one. Code hoisting inserts the unified expression in a common *dominator* location, while code sinking places it in a common *post-dominator* location.

2.2 Branch Fusion

Coutinho *et al.* [10] originally proposed an optimization called *branch fusion* in order to reduce control-flow divergence in GPU kernels and improve performance. Their technique works on *diamond-shaped if-then-else* constructs, where each of the two paths contains a single basic block. For these two blocks, they align and merge equivalent instructions using sequence alignment. Instructions with the same opcodes and data types are matched and merged, potentially using extra conditional select statements to handle argument mismatches. Otherwise, the execution of mismatched instructions is controlled through *if-then-else* constructs on the same condition as the original branch. While code motion is based on value equivalence, branch fusion is based only on opcode and type equivalence. However, while code motion is able to move any amount of copies into a common location, branch fusion is limited to merging only pairs of instructions, *i.e.,* one from each side of the conditional branch.

Our central insight is that branch fusion can also be employed for code size reduction. Figure 1 shows an example of branch fusion in C code extracted from the Linux kernel. Traditional local code motion optimizations, such as hoisting and sinking, cannot merge the statements in the two paths of the *if-then-else*. While both contain very similar calls to `rcu_btrace`, but they are not equal in value, which is what determines whether hoisting and sinking can be applied. On the other hand, branch fusion merges code paths based on

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

```
if (atomic_dec_test(&rcus.bcpucount)) {
  rcu_btrace(TPS("LastCB"), -1, rcus.bseq);
  complete(&rcus.bcompletion);
} else {
  rcu_btrace(TPS("CB"), -1, rcus.bseq);
}
```

(a) Before the state-of-the-art branch fusion.

```
1 cond = atomic_dec_test(&rcus.bcpucount);
2 str = cond?"LastCB":"CB";
3 rcu_btrace(TPS(str), -1, rcus.bseq);
4 if (cond) {
5   complete(&rcus.bcompletion);
6 }
```

(b) After the state-of-the-art branch fusion.

Figure 1. Code extracted from `rcu_barrier_callback` in the Linux kernel. Matching statements are highlighted in green, and mismatched (sub-)statements in red. In the fused code (b), the mismatched arguments are handled with a select statement (line 2), while the mismatched statement (line 5) with an *if* on the original condition. This results in a code size reduction of 18 bytes, or 11%.

their *operation equivalence*. Value differences are handled via conditional value selection. Figure 1(b) shows the code after branch fusion is applied.

2.3 Limitations of the State of the Art

A recent work has introduced DARM [31], a branch fusion technique that exploits structural similarity in *single-entry single-exit* (SESE) regions. Similar to Coutinho *et al.* [10], DARM also aims at reducing thread divergence in GPU kernels. However, exploiting structural similarity is quite important in reducing code size too. To further motivate this, consider the code shown in Figure 2. This code contains an *if-else-if* statement both cases of which call the same macro `DeleteNode` (highlighted in Figure 2). Macro expansion causes the branch to have isomorphic regions with highly similar code, exactly the kind of code DARM handles efficiently.

Although DARM is a significant improvement over its predecessor, we have identified two major limitations with regards to code size reduction: first, it is limited to SESE regions; second, it depends on the two regions having structural similarity. Figure 3 shows an example extracted from the PostgreSQL database system [24], in the `text_substring` function. None of the existing branch fusion techniques can merge the two paths of the *if-then-else*: the original branch fusion technique is limited to *if-then-else* constructs with a single basic block on each side, while DARM can only handle SESE regions. In contrast, each side of the *if-then-else* construct in the example is a *single-entry multiple-exit* (SEME) region: some statements transfer control to the end of the *if-then-else*, while the return statements transfer control outside the function. Even if we unify all the return instructions (a common transformation), the code still has two distinct exit points which cannot be handled by either of the existing

```
#define DeleteNode(x) { \
  xx_hold = (x); \
  while ( Up(xx_hold) != xx_hold ) \
    DeleteLink( Up(xx_hold) ); \
  while ( Down(xx_hold) != xx_hold ) \
    DeleteLink( Down(xx_hold) ); \
  Dispose(xx_hold); \
}
if ( prnt_flush ) {
  Parent(prnt, Up(dest_index));
  if ( kill )  DeleteNode(dest_index);
  debug0(DGF, DD, "calling FlushGalley...");
  FlushGalley(prnt);
} else
  if ( kill )  DeleteNode(dest_index)
```

Figure 2. Code snippet extracted from file `z20.c` in MiBench `typeset` benchmark

branch fusion techniques. Despite that, there is significant potential in merging the two regions. The *true-side* and the *false-side* regions have 7 and 31 basic blocks, respectively, out of which, 5 pairs (10 blocks in total) can be perfectly aligned. Overall, 24.4% of the LLVM instructions are mergeable.

The second major limitation that we have identified in DARM concerns its restriction on merging only regions with structural similarity (*i.e.,* SESE regions with isomorphic control-flow graphs). Regions with non-isomorphic control-flow graphs can still have enough code similarity to profitably merge them. Therefore, we propose SEME-Fusion a technique that can merge SEME regions without the isomorphism requirement.

Another important observation is that none of DARM or SEME-Fusion can be the best fusion technique for given CFG. SEME-Fusion works best for the code shown in Figure 3, but DARM can extract the most similarity in Figure 2 because its ability to match structurally similar regions works into its advantage in that case. Therefore, we believe a combination of these two techniques is needed to obtain the best of both worlds. In the next sections: We describe how we adapt and improve DARM for code reduction; We describe a novel branch fusion technique that addresses the limitations of DARM; Finally, we offer a branch fusion framework that combines these techniques, taking advantage of all their individual strengths.

3 A Branch Fusion Framework for Code Size Reduction

In this section, we introduce *HyBF*, a framework for code size oriented branch fusion. The proposed framework reduces code size using a three step process. First, we find conditional branches and we collect their two disjoint regions that represent the *then* and the *else* paths. Second, we attempt to merge the two regions using one or more branch fusion techniques. Here we consider the following two novel techniques:

```
if (eml == 1) {
 S1 = Max(S, 1);
 if (length_not_specified)
  L1 = -1;
 else {
  int  E = S + length;
  if (E < S)
   ereport(ERROR, errcode, errmsg);
  if (E < 1)
   return cstring_to_text("");
  L1 = E - S1;
 }
 return DatumGetTextPSlice(str, S1-1, L1);
} else {
 if (eml > 1) {
  S1 = Max(S, 1);
  slice_start = 0;
  if (length_not_specified)
   slice_size = L1 = -1;
  else {
   int E = S + length;
   if (E < S)
    ereport(ERROR, errcode, errmsg);
   if (E < 1)
    return cstring_to_text("");
   L1 = E - S1;
   slice_size = (S1+L1)*eml;
  }
  ...
  if (slice!=(text*)DatumGetPointer(str))
   pfree(slice);
  return ret;
 } else
  elog(ERROR, "invalid encoding");
}
```

Figure 3. Code extracted from `text_substring` in the PostgreSQL database system. Matching statements are highlighted in green, and mismatched (sub-)statements in red. This *if-then-else* example contains two SEME regions.

- **Control-flow melding for code size (CFM-CS)** that focuses on merging isomorphic SESE regions.
- **Single-entry-multiple-exit branch fusion (SEME-Fusion)** that can merge SEME regions without the restriction of structural similarity.

The best approach for each specific conditional branch depends on the CFG and the instructions in the two paths of the branch. CFM-CS may work best for branches with highly similar code structure, while SEME-Fusion is preferred when the two sides of the branch have multiple exits and/or low structural similarity.

The third and final step is to apply both branch fusion techniques on every candidate branch. We perform a profitability analysis using a code-size cost model to determine whether any technique results in smaller code than the original *if-then-else* construct. If both reduce the code size, we keep the output of the best one and discard the other.

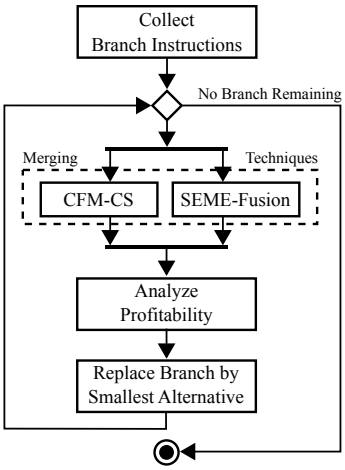
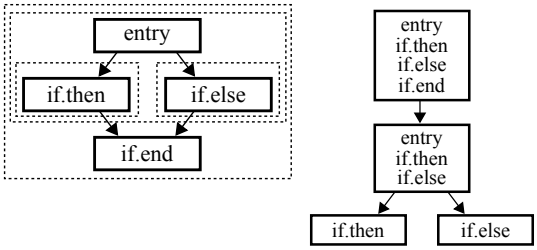


Figure 4. Overview of the *HyBF* framework.



(a) A control-flow graph with its regions highlighted. **(b)** Region hierarchy graph.

Figure 5. A simple control-flow graph and its region hierarchy graph.

3.1 Searching for Conditional Branches

Given an input function, we search for conditional branches that represent an *if-then-else* construct. For each conditional branch, we collect their two disjoint regions that represent the *then* and the *else* paths. Different branch fusion techniques considered in this paper have different limitations on which regions they are capable of merging. State-of-the-art branch fusion [10] can merge only single basic blocks. CFM-CS, which we describe in more detail in Section 4, works on single-entry single-exit (SESE) regions. SEME-Fusion, described in Section 5, works on single-entry multiple-exit (SEME) regions. Therefore, we use different filtering strategies for each technique we explore in this paper.

Since both SESE and SEME regions can themselves include other conditional branches, the traversal order may also affect the end result. In both cases, we can construct a region hierarchy graph, as illustrated in Figure 5, and decide between either a top-down or a bottom-up traversal. A bottom-up traversal starts from the inner-most regions and progress to the outer-most ones, while the top-down traversal goes from the outer to the inner-most regions. While smaller regions are more likely to contain greater similarities, merging larger regions can result in larger reductions.

3.2 Profitability Analysis

When merging two non-identical regions from a conditional branch, the extra code necessary for handling their differences might result in an overall code increase. Therefore, we need to identify opportunities where branch fusion is profitable and reduces code size. To this end, we employ a profitability analysis. The final decision is based on estimates of the code-size costs for both the merged and the original *if-then-else*. The version with the smallest estimated code size is chosen.

The profitability is measured with the help of the compiler's target-specific cost model. The cost of each instruction comes from querying the compiler's built-in cost model, which provides a cost estimation that approximates the size of an IR instruction when lowered to the target machine. We use the code-size cost model provided by LLVM's target-transformation interface (TTI), which is used in the decision making of most optimizations [30, 40].

4 Control-Flow Melding

Control-flow Melding [31] (DARM) is a code optimization technique used for reducing control-flow divergence in GPU programs. DARM reduces divergence by merging similar control-flow regions contained within divergent branches of the CFG. Previous compiler-based divergence reduction techniques such as Tail Merging and Branch Fusion are unable to merge control-flow beyond basic block boundaries. Therefore, they have limited applicability in real-world programs. DARM was proposed to fill this gap and enable merging control-flow at region level. DARM works by merging structurally similar (*i.e.,* isomorphic) single-entry single-exit (SESE) regions within *if-then-else* branches. Even though the general idea of merging similar control-flow regions is applicable to real-world programs, DARM's implementation is fairly restrictive as it only supports merging simple nested *if/if-else* statements and *loops* inside *if-then-else* branches.

In this work, we extend and adapt DARM to reduce code size in CPU programs. In the following sections we describe the main steps in Control-flow Melding for Code Size Reduction (CFM-CS). Figure 6 shows the main stages of CFM-CS.

4.1 Identifying Regions for Melding

The first step of CFM-CS is identifying on which locations to apply the transformation. As described in Section 3.1, CFM-CS is applicable to *if-then-else* constructs that contains isomorphic control-flow regions. To formally describe the conditions that a valid location must satisfy, consider the CFG in Figure 6 ⓐ. This CFG contains a basic block E with a conditional branch at its end. Basic blocks L and R be the two successors of E. Let X be the immediate post-dominator of E. E dominates all basic blocks contained within the SESE region E-X. E is considered to be a valid location for our

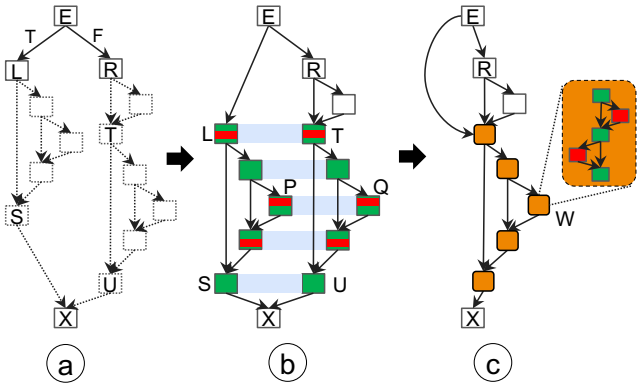


Figure 6. CFM-CS overview. (a) Given an *if-then-else* statement, (b) we identify isomorphic control-flow in the two regions, and (c) we align and merge the corresponding blocks.

transformation if there exist no paths in the CFG from E to X that goes through both L and R. This ensures that either L or R is executed at a time but not both, enabling us to *at least* merge the common computations within L and R. If there exists a path from L to R at least one predecessor of R must be dominated by L because all program paths from E to X must go through either L or R. We use this property to check non-existence of paths from L-R or R-L. In addition, basic blocks contained within E-X must not contain unhandled instructions for CFM-CS to be applicable[2].

The next step of CFM-CS is to collect all the subregions contained with the parent region of E-X. We employ LLVM's region tree (*i.e.,* region hierarchy graph) [17] data structure to do this. We collect subregions along the left path (from L to X) and right path (from R to X). Each subregion is selected such that subregion entry is dominated by L or R and subregion exit post-dominates L or R. For example, the CFG in Figure 6 ⓐ has the subregion L-S on left path and subregions R-T,T-U on the right path. Any isomorphic SESE subregion pair consisting of one subregion from left and right paths can be merged to potentially reduce code size. We use a heuristic-based approach based on instruction frequencies and their size cost to determine what isomorphic subregion pairs to merge. Isomorphic SESE subregions with more similar instructions are more profitable to be merged together. We formulate this as a sequence alignment problem and solve it using the Smith-Waterman algorithm [34]. For example, in Figure 6 ⓑ isomorphic subregions L-S and T-U are aligned together and their corresponding basic blocks (shown connected with light blue bars) can be merged.

[2]Even though this is not a strict limitation of CFM-CS transformation, we do not merge regions containing switch-case constructs

4.2 CFM-CS Code Generation

We compute an instruction alignment similar to DARM [31] or HyFM [28], to generate the final merged regions. Instruction alignment is computed for each corresponding basic block pair in aligned subregions. In Figure 6 ⓑ portions of basic blocks with perfectly aligned instructions are shown in green and unaligned portions are shown in red. We use instruction alignment to generate the final merged code (shown in Figure 6 ⓒ). The aligned instructions are replaced with merged instructions that use *select* instructions to pick their operands, while the unaligned instructions are moved to new basic blocks and executed conditionally. For example, matched basic blocks P and Q have both aligned and unaligned portions and the final merged CFG for these blocks are shown zoomed-in on Figure 6 ⓒ. Note that the orange colored blocks are not necessarily basic blocks but control-flow subgraphs. We use the branching condition at block E as the distinguishing predicate for the *select* operations as well as for conditionally executing unaligned instructions.

4.3 Region Replication

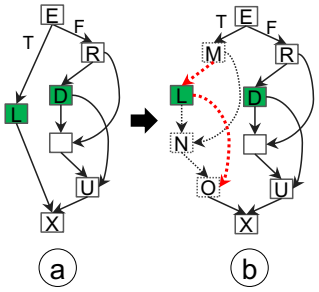


Figure 7. Region replication example

CFM-CS is only capable of merging isomorphic regions, however there is one exception. When one path contains only a single basic block and the other path contains region(s), CFM-CS can still be applied by using *Region Replication*. The idea here is to replicate a region and place the single basic block in a convenient location to enable profitable merging. Region replication was first proposed in DARM [31] to merge basic blocks in *if-else-if* chains to reduce control-flow divergence. Their implementation did not support more complex control-flow patterns because the primary focus was to reduce divergence in some select control-flow patterns seen in GPU kernels. We build on top of DARM and provide a more general region replication approach that is applicable to code size reduction in CPU programs. We use example CFGs shown in Figure 7 to explain how region replication works. The input CFG (block L in Figure 7 ⓐ) contains a single basic block on the left path and a subregion in the right path. Assume that the computations done in blocks L and D are similar and merging them is profitable. First we replicate the right subregion R-U and create a new subregion M-O (Figure 7 ⓑ). And then we place L on a corresponding position to D. This creates two isomorphic regions and we can apply CFM-CS region merging approach[3]. We also make sure values produced in L will reach their external users by inserting ϕ nodes at L's new dominance frontiers (in this case L has two dominance frontiers N and O). We concretize the path conditions on region M-O to make sure L is always executed (concretized path $M \rightarrow L \rightarrow O$ is shown in red) and also make sure phi-nodes in block X pick the correct incoming values based on the chosen path.

5 Branch Fusion for SEME Regions

CFM-CS uses techniques inspired by function merging [28] to align and merge already matched basic blocks. A more straightforward approach is to generalize function merging and apply it directly on the two paths of an *if-then-else* statement. This section explains this novel branch fusion approach, *SEME-Fusion*. It expands the state-of-the-art function merging technique [28, 36], which uses an alignment strategy for identifying similarities.

5.1 Extracting SEME Regions from Branches

For each conditional branch, we need to identify if it represents a valid *if-then-else* construct. To this end, we collect the two maximal SEME regions starting from the two successors of the conditional branch, following the *then* and *else* paths. A SEME region is a control-flow graph with a single incoming edge where the entry block dominates all other basic blocks in the region. The SEME region is *maximal* if no external block adjacent to the SEME region is also dominated by the entry block.

5.2 Merging Two SEME Regions

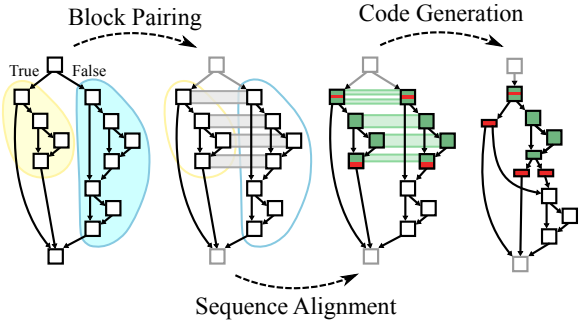


Figure 8. Overview of the SEME-Fusion technique.

Since a function is also a SEME region, we can leverage existing function merging techniques, adapting them for

[3]Alternatively, blocks L and D in Figure 7 ⓐ can be merged directly and direct jumps from $E \rightarrow D$ and $D \rightarrow X$ can be inserted to ensure correct control. CFM-CS does not use this approach because applying it recursively can make the CFG more complex/unstructured and unamenable to other optimizations including CFM-CS

branch fusion [28–30, 36]. Our merging technique is based on the HyFM function merging technique [28], which can efficiently merge any two functions. Figure 8 shows an overview of our novel branch fusion technique.

Our merging technique works at the basic-block level. Given a conditional branch and its two SEME regions, first we generate a fixed-vector representation for all their basic blocks, namely, their fingerprint [29, 30]. We pair similar basic blocks across the two SEME regions based on their fingerprint distances. For each selected pair of basic blocks, we use linear pairwise alignment [28] to identify matching segments of equivalent instructions across the pair.

Then, we employ the first-tier profitability analysis on the resulting alignment to decide whether this pair of basic blocks will be merged or not. The first-tier profitability analysis consists of a simple analysis applied on each pair of basic blocks selected for alignment. If the cost model deems it unprofitable, we discard this pair and its alignment, freeing the basic blocks to be paired with others. The pairing of basic blocks, the alignment, and the first-tier profitability analysis are executed in rounds, in a greedy manner.

Once all basic blocks have been processed, we use the resulting alignments to generate the code of the merged region: aligned instructions are replaced by a single instruction in the merged code, while blocks of unaligned instructions are copied as they are, but with their execution conditional on the *if-then-else* condition. If no pair of basic blocks with a profitable alignment was found, we bail out early.

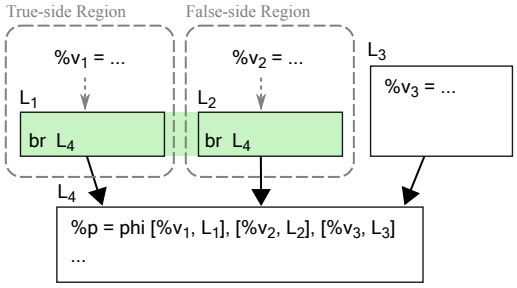
5.3 Adjusting Phi-Nodes in the Exit Blocks

After branch fusion, we need to adjust the list of incoming blocks of the phi-nodes in the exit blocks. Incoming values from basic blocks outside the two regions are kept as they are. For incoming values from either one of the original SEME regions, in the general case, we simply remap them to their corresponding values in the merged region, where their incoming basic blocks are also remapped accordingly.

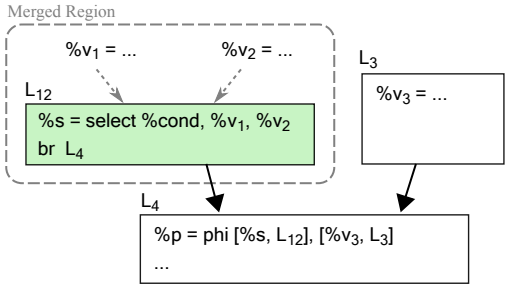
Figure 9 illustrates a special case. If a phi-node of an exit block has two incoming basic blocks that have been merged, we also need to merge the two corresponding incoming values, as shown in Figure 9b. Since one incoming block is from the *true-side region* of the conditional branch and the other is from the *false-side region*, we can use the same branch condition to select between the two incoming values. Therefore, we simply add a `select` instruction in the merged block to select between the two corresponding values depending on the branch condition. Note that same strategy is used in CFM-CS when merging regions exits that have a common successor that includes phi-nodes.

6 Evaluation

In this section, we evaluate our branch fusion framework and the different merging techniques. We implemented *HyBF*



(a) The original code before branch fusion.



(b) After branch fusion, value selections are added to the merged incoming blocks.

Figure 9. Example illustrating how a phi-node in an exit block must be adjusted when its incoming blocks are merged.

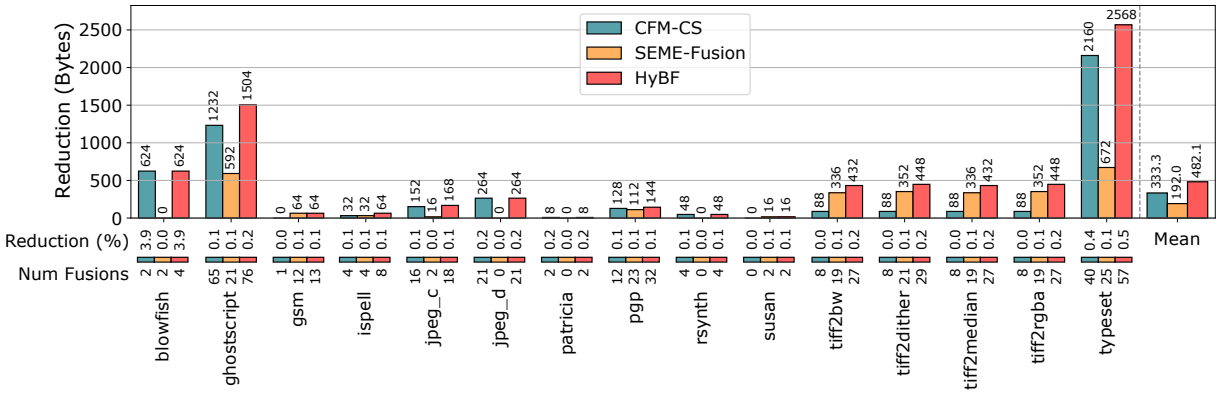
and its two component techniques, CFM-CS from Section 4 and SEME-Fusion from Section 5 as LLVM transformation passes[4]. *HyBF* can be implemented on any static single assignment (SSA) [11] based intermediate representation and does not depend on any LLVM-IR specific feature to the best of our knowledge. We apply our transformations together with `-Oz`. We place branch fusion after the classic redundancy elimination and code motion passes, as they can be negatively affected by branch fusion. We also evaluate the gains achieved by branch fusion on top of the state-of-the-art function merging [28, 36] technique in LTO mode. We evaluate *HyBF*, the two individual techniques, and the no-fusion baseline on four different benchmark suites: AnghaBench [12], MiBench [16], SPEC 2006, and SPEC 2017 [35]. These benchmarks cover a variety of applications including compilers, interpreters, typesetting, 3D rendering, and cryptography.

We perform all experiments on a server with two octa-core Intel Xeon E5-2650 processors and 64 GiB of RAM, running Ubuntu 18.04.3 LTS. For timing measurements, we repeat all experiments 10 times to minimize the effect of measurement noise. We test whether different sets of measurements are statistically indistinguishable using the t-test with $p < 5\%$.

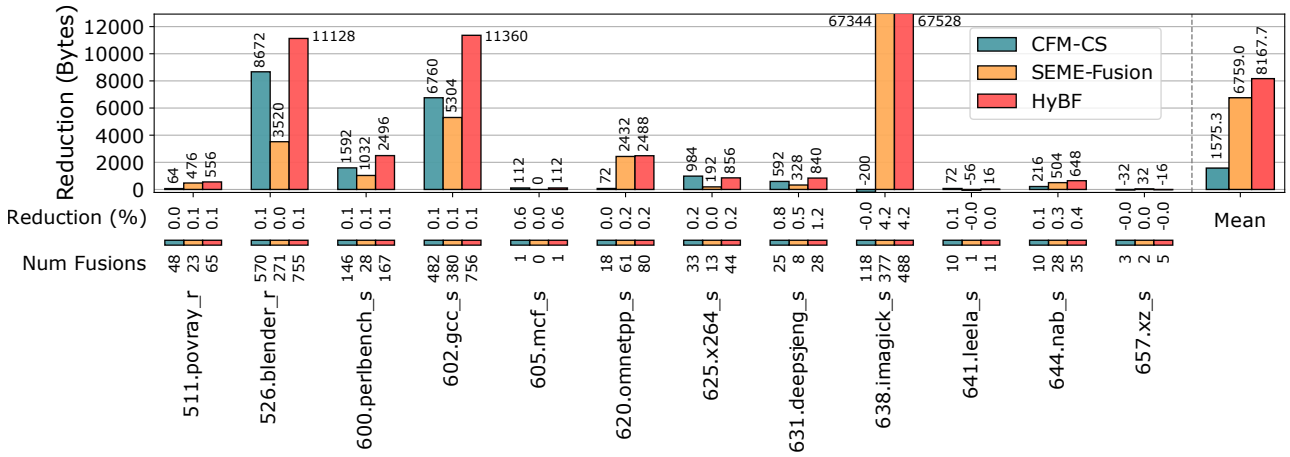
6.1 Code Size Reduction

Figure 10 shows the code size reduction achieved by the different techniques on the MiBench and SPEC 2017 benchmark suites. We show both the absolute and the relative

[4]LLVM-14.0



(a) *HyBF* reduces a total of 7.1 KB on top of `-Oz` across all MiBench programs.



(b) *HyBF* reduces a total of 95.7 KB on top of `-Oz` across all SPEC 2017 programs.

Figure 10. Code size reduction on full programs from MiBench and SPEC 2017. Top part of each sub-figure shows the reduction (relative to the baseline) in bytes. Below that is the relative reduction in %. At the bottom is the number of profitable fusion operations applied. *HyBF* reduces code size in 43 out of the 61 benchmarks considered, achieving up to 4.2% code size reduction.

reduction in the *text* section of the final binary over the baseline. We also show how many branch fusion operations were successfully performed by each version, according to the profitability analysis.

Across all three suites, it is clear that neither CFM-CS nor SEME-Fusion suffices by itself. CFM-CS does better for 22 benchmarks but SEME-Fusion outperforms it for 20. Using only one of the two techniques is suboptimal. *HyBF* allows us to pick and choose the best one for each individual branch, as determined by the compiler's cost model. As a result, *HyBF* outperforms both techniques.

There are a few corner cases where this is not true. An example is `456.hmmer` where SEME-Fusion has a small gain over *HyBF*. For branches that can be fused by both CFM-CS and SEME-Fusion, *HyBF* relies on the compiler's cost model to decide which one produces the smallest merged code. Due to inaccuracies of the cost model, some of these decisions can be sub-optimal.

On MiBench, a benchmark suite for embedded systems, CFM-CS was the best individual technique, reducing code size by 333 bytes on average, while SEME-Fusion achieved an average reduction of 192 bytes. *HyBF* combined the best cases of the two techniques, achieving an average reduction of 482 bytes. Unsurprisingly, the two largest MiBench programs, namely, `ghostscript` and `typeset`, with 3446 and 362 functions, respectively, also have the highest number of successful branch fusions.

On SPEC 2017, SEME-Fusion was the best individual technique, achieving an average reduction of 6759 bytes compared to 1575.3 bytes by CFM-CS alone. Again, *HyBF* combines their best cases, reducing code size by 8167.7 bytes on average. The best case is for `638.imagick_s`, where *HyBF* applies 488 successful branch fusion operations and reduces size by 67 KB, a 4.2% reduction compared to the *no-fusion* baseline. This is an interesting example because CFM-CS and SEME-Fusion have very different effects: CFM-CS alone *increases* size by 200 bytes, with almost all the code size reduction coming from SEME-Fusion. Again, *HyBF* outperforms both which shows that choosing the best technique

on a branch-by-branch basis is the superior approach. Overall, SEME-Fusion outperforms CFM-CS and *HyBF* extracts the best of both techniques.

We have also evaluated on SPEC 2006, where the results follow a very similar pattern to SPEC 2017. *HyBF* achieves an average reduction of 1760 bytes, outperforming both techniques individually. As in the case of SPEC 2017, on average, SEME-Fusion outperforms CFM-CS.

There are also some cases where branch fusion leads to some small code degradation, where inaccuracies in the cost model mislead the profitability analysis. These are common when there are only a few successful branch fusion operations, as shown in Figure 10. For programs with several successful branch fusion operations, large gains tend to outweigh some small losses caused by misjudgments of the profitability analysis.

6.1.1 Effectiveness on Real-World Code. We also test the effectiveness of branch fusion on real-world code using the AnghaBench suite [12]. AnghaBench provides one million compilable functions which were extracted from the most popular GitHub repositories with C source files. This includes well-known repositories such as PostgreSQL, numpy, Linux, FFmpeg, etc.

Figure 11 shows how well we do on real-world code. It plots the relative reduction achieved by CFM-CS and *HyBF* in terms of LLVM-IR instructions for the 17.6k functions that are visibly affected by at least one of the techniques. Beyond achieving significant code size reduction, up to 60%, the plot highlights again the importance of using different branch fusion techniques in a coordinated way. CFM-CS alone finds several opportunities but in most cases, *HyBF* either matches or surpasses it. For almost 6500 functions where CFM-CS has no impact, *HyBF* finds enough opportunities, in some cases reducing their size by up to 50%.

There are also cases where branch fusion leads to code increase. As previously discussed, these regressions stem from inaccuracies in the cost model. The main one has to do with the cost of phi-nodes. To handle the merged control flows, our branch fusion approaches often add a significant number of extra phi-nodes, even in otherwise small functions. Because phi-nodes usually have a small cost, compared to other instruction opcodes at least, branch fusion may lead to an overall increase in the number of LLVM-IR instructions. For small functions, this result in a relatively large regressions.

6.1.2 Comparison Between CFM-CS and SEME-Fusion. In full programs, there are several benchmarks where CFM-CS outperforms SEME-Fusion. While the reduction in binary size is a result of applying the merging transformations at multiple locations in multiple source files in these benchmarks, we found several interesting functions where CFM-CS does clearly better than SEME-Fusion. One such case is shown in Figure 2 (Section 2). As described in Section 2.3, this function happens to have isomorphic code regions with

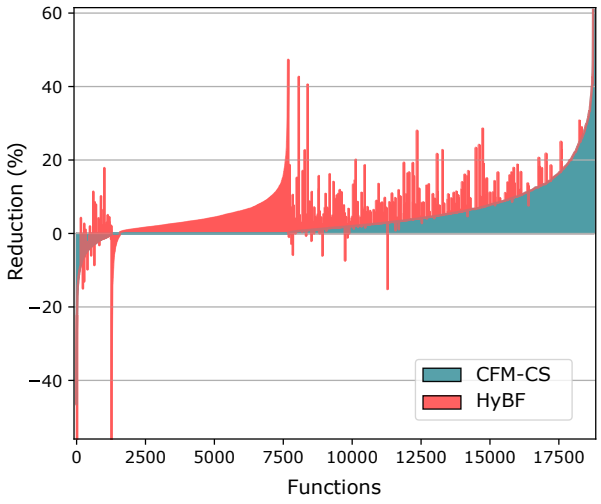


Figure 11. LLVM-IR size reduction for CFM-CS and *HyBF* on 17.6k real-world functions from AnghaBench. x-axis is function id and is sorted by reduction for CFM-CS.

highly similar code and applying CFM-CS results in LLVM-IR size reduction of 29%.

SEME-Fusion is also applicable on this branch but it cannot pair the identical CFGs directly. Its pairing strategy is basic block based and does not take the overall CFGs into account. Moreover, when using the pairwise alignment, as proposed by HyFM [28], SEME-Fusion can only pair basic blocks with the same number of instructions. In this case, the matched blocks are not the equivalent blocks in the two identical CFGs. This results in unnecessary overhead for handling the divergent control flow inside the merged code, including operand selections, phi-nodes, and branches. This extra complexity renders the merged code unprofitable.

However, there are also several cases where SEME-Fusion outperforms CFM-CS. As discussed in Section 2.3, DARM and therefore CFM-CS, has several limitations that are specifically addressed by SEME-Fusion. First, for regions with multiple blocks but no structural similarity, CFM-CS is insufficient, while SEME-Fusion can still identify pairs of basic blocks that are worth merging between those two regions. Second, SEME-Fusion is the only technique capable of merging branches with SEME regions. In fact, only SEME-Fusion could have merged the code shown in Figure 3, reducing the size of the IR of this function by 25 instructions and of the final binary by 101 bytes.

The major benefit of *HyBF* is that it can take advantage of the different strengths of both CFM-CS and SEME-Fusion, whichever suits each branch best.

6.1.3 Comparison with Function Merging. Function merging is an important code size optimization, where recent work has shown significant reduction gains [26, 28]. In this section, we analyze the interaction between branch fusion and function merging. While function merging works

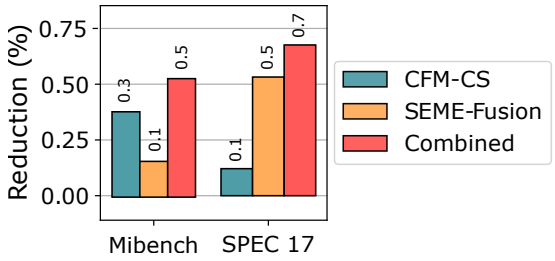


Figure 12. Average code size reduction on top of the state-of-the-art function merging on LTO mode.

at the function level as an inter-procedural optimization, branch fusion works at the region level within single functions. Although the two optimizations are complementary to each other, the extra complexity added by branch fusion to regions of a given function can reduce its similarity to other functions, affecting function merging.

Our evaluation shows that *HyBF* is able to offer additional reduction gains on top of function merging while having a negligible impact on the total number of merged functions. Across the SPEC 2017 benchmark suite, HyFM achieves a total of 8194 profitable merge operations with *HyBF* disabled versus 8180 with it enabled, *i.e.,* this is a reduction of 0.2% on the total number of merged functions. On the other hand, across all SPEC 2017 programs, *HyBF* achieves a total of 2898 profitable branch fusion operations, reducing a total of 253.4 KB on top of the already highly optimized binaries with HyFM in LTO mode. This represents a significant gain if we compare to enabling only either CFM-CS or SEME-Fusion, where we achieve a total of 59.4 KB or 192.4 KB, respectively. Figure 12 shows the average percentage reduction achieved on top of HyFM across all programs of each benchmark suite.

7 Related Work

Compiler-based code size reduction is important for fitting large programs to resource-constraint embedding devices. Previous approaches reduce code size by replacing a code segment with a smaller, semantically-equivalent implementation [23, 38], deleting unnecessary code [9, 19], combining redundant code within a function [5, 8] or across functions [4, 22]. Function merging falls into the later category.

Established compilers like GCC and LLVM [1, 21] provide an optimization for merging identical functions at the IR level. They can only handle type mismatches that can be losslessly cast to the same format. Von Koch et al. [14] extended this idea into merging nearly identical functions. They restrict merging to functions with the same signature, and identical control-flow graphs. In addition, corresponding blocks of the functions must have the same number of instructions. Paired instructions are allowed to have different opcodes or list of arguments but must have equivalent data types.

Rocha et al. [29, 30] was the first to propose a code-size optimization capable of merging arbitrary pairs of functions. They employ a sequence alignment algorithm to find equivalent code segments that can be merged into a single function, while the mismatching segments of code are also added to the merged function but have their code guarded by a function identifier. More recently, they have proposed HyFM [28, 37], where they avoid the quadratic aspect of the alignment operation in two folds: first, HyFM works on the basic block level, reducing the granularity of the inputs for the alignment algorithm in practice; second, it employs a simpler linear alignment strategy.

Loop rolling is another important code-size optimization that works by merging similar code into a loop [15, 26]. The state-of-the-art technique, RoLAG [26], searches for isomorphic instructions within a single basic block and builds an alignment graph representing the groups of isomorphic instructions that can be rolled into a loop. While both loop rolling and branch fusion work within functions, they exploit different opportunities for merging similar code, since the former considers similarities within a basic block while branch fusion consider similarities across the regions that form an *if-then-else* construction.

Chen *et al.* proposed *Generalized Tail Merging* for code size reduction [6]. This extends the tail merging to work on isomorphic SEME regions. Although conceptually similar to CFM-CS, this technique does not use instruction alignment and requires instructions inside the SEME regions to be nearly identical. This restricts the applicability of generalized tail merging to a subset of cases that CFM-CS can handle. In contrast, *SEME-Fusion* does not require structural similarity both these techniques require.

8 Conclusion

Existing optimizations such as tail merging, branch fusion, code hoisting/sinking only merge instructions on branch paths that contain single basic blocks. Therefore, they have limited applicability in real-world programs. We observe that in many applications there are plenty of opportunities to merge branch paths that contain multiple basic blocks. We propose *HyBF*, a new hybrid approach for code size reduction at conditional branches. *HyBF* incorporates two new techniques, CFM-CS that merges isomorphic SESE regions and SEME-Fusion that can merge SEME regions within conditional branches. Our evaluation shows that *HyBF* achieves the best of both worlds by picking the best technique for a given program location to maximally reduce code size.

Data-Availability Statement

All the code and scripts necessary for replicating the main experiments in the paper are distributed in a publicly available artifact [25].

Acknowledgment

This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] 2020. The LLVM Compiler Infrastructure. MergeFunctions pass, how it works. http://llvm.org/docs/MergeFunctions.html.

[2] Rafael Auler, Carlos Eduardo Millani, Alexandre Brisighello, Alisson Linhares, and Edson Borin. 2017. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform. *Concurrency and Computation: Practice and Experience* 29, 22 (2017), e3932. https://doi.org/10.1002/cpe.3932

[3] Preston Briggs and Keith D. Cooper. 1994. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 159–170.

[4] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-size Optimization for Production iOS Mobile Applications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, US, 1–12. https://doi.org/10.1109/CGO51591.2021.9370306

[5] Wen Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code Compaction of Matching Single-Entry Multiple-Exit Regions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–417. https://doi.org/10.1007/3-540-44898-5_23

[6] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code Compaction of Matching Single-Entry Multiple-Exit Regions. In *Proceedings of the 10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 401–417.

[7] Cliff Click. 1995. Global Code Motion/Global Value Numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/207110.207154

[8] John Cocke. 1970. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization*. ACM, New York, NY, USA, 20–24. https://doi.org/10.1145/800028.808480

[9] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, USA) *(LCTES '99)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/315253.314414

[10] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira Jr. 2011. Divergence Analysis and Optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 320–329.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. ACM, New York, NY, USA, 25–35.

[12] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[13] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (mar 2000), 378–415. https://doi.org/10.1145/349214.349233

[14] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. Exploiting Function Similarity for Code Size Reduction. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/2666357.2597811

[15] Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. 2022. RollBin: Reducing Code-Size via Loop Rerolling at Binary Level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, CA, USA) *(LCTES 2022)*. Association for Computing Machinery, New York, NY, USA, 99–110. https://doi.org/10.1145/3519941.3535072

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

[17] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.* 29, 6 (June 1994), 171–185. https://doi.org/10.1145/773473.178258

[18] S. L. Keoh, S. S. Kumar, and H. Tschofenig. 2014. Securing the Internet of Things: A Standardization Perspective. *IEEE Internet of Things Journal* 1, 3 (June 2014), 265–275. https://doi.org/10.1109/JIOT.2014.2323395

[19] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/773473.178256

[20] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: Inter-Procedural Basic Block Layout Optimization. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 65–75. https://doi.org/10.1145/3302516.3307358

[21] Martin Liška. 2014. Optimizing large applications. *arXiv preprint arXiv:1403.6997* (2014).

[22] Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes. 2004. Code factoring in GCC. In *Proceedings of the 2004 GCC Developers' Summit*. 79–84.

[23] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36177.36194

[24] Behandelt PostgreSQL. 1996. PostgreSQL. *Web resource: http://www.PostgreSQL.org/about* (1996).

[25] Rodrigo Rocha, Pavlos Petoumenos, Charitha Saumya, and Kirshanthan Sundararajah. 2023. [Artifact] HyBF : A hybrid branch fusion strategy for code size reduction. (1 2023). https://doi.org/10.6084/m9.figshare.21976358.v1

[26] Rodrigo C. O. Rocha, Pavlos Petoumenos, Bjorn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[27] Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[28] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function Merging for Free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Virtual, Canada) *(LCTES 2021)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/3461648.3463852

[29] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code*

Generation and Optimization (CGO 2019). IEEE Press, Piscataway, NJ, USA, 149–163. https://doi.org/10.1109/CGO.2019.8661174

[30] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868. https://doi.org/10.1145/3385412.3386030

[31] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2022. DARM: Control-Flow Melding for SIMT Thread Divergence Reduction. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 28–40. https://doi.org/10.1109/CGO53902.2022.9741285

[32] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. 2003. Compiling Java for Low-end Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (San Diego, California, USA) *(LCTES '03)*. ACM, New York, NY, USA, 42–50. https://doi.org/10.1145/780732.780739

[33] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder. 2012. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine* 50, 12 (December 2012), 144–149. https://doi.org/10.1109/MCOM.2012.6384464

[34] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195 – 197.

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

[36] Sean Stirling, Rocha Rodrigo C. O., Kim Hazelwood, Hugh Leather, Michael O'Boyle, and Pavlos Petoumenos. 2022. F3M: Fast Focused Function Merging. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 242–253. https://doi.org/10.1109/CGO53902.2022.9741269

[37] Sean Stirling, Rocha Rodrigo C. O., Kim Hazelwood, Hugh Leather, Michael O'Boyle, and Pavlos Petoumenos. 2022. F3M: Fast Focused Function Merging. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 242–253. https://doi.org/10.1109/CGO53902.2022.9741269

[38] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. 1982. Using Peephole Optimization on Intermediate Code. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 21–36. https://doi.org/10.1145/357153.357155

[39] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[40] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. 2018. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. ACM, New York, NY, USA, 12:1–12:15.

[41] A. Varma and S. S. Bhattacharyya. 2004. Java-through-C compilation: an enabling technology for Java in embedded systems. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 3. 161–166 Vol.3. https://doi.org/10.1109/DATE.2004.1269224

[42] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

Received 2022-11-10; accepted 2022-12-19