Effective Function Merging in the SSA Form

Rodrigo C. O. Rocha
University of Edinburgh, UK
r.rocha@ed.ac.uk

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

Zheng Wang
University of Leeds, UK
z.wang5@leeds.ac.uk

Murray Cole
University of Edinburgh, UK
mic@inf.ed.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Abstract

Function merging is an important optimization for reducing code size. This technique eliminates redundant code across functions by merging them into a single function. While initially limited to identical or trivially similar functions, the most recent approach can identify all merging opportunities in arbitrary pairs of functions. However, this approach has a serious limitation which prevents it from reaching its full potential. Because it cannot handle phi-nodes, the state-of-the-art applies register demotion to eliminate them before applying its core algorithm. While a superficially minor workaround, this has a three-fold negative effect: by artificially lengthening the instruction sequences to be aligned, it hinders the identification of mergeable instruction; it prevents a vast number of functions from being profitably merged; it increases compilation overheads, both in terms of compile-time and memory usage.

We present SalSSA, a novel approach that fully supports the SSA form, removing any need for register demotion. By doing so, we notably increase the number of profitably merged functions. We implement SalSSA in LLVM and apply it to the SPEC 2006 and 2017 suites. Experimental results show that our approach delivers on average, 7.9% to 9.7% reduction on the final size of the compiled code. This translates to around 2× more code size reduction over the state-of-the-art. Moreover, as a result of aligning shorter sequences of instructions and reducing the number of wasteful merge operations, our new approach incurs an average compile-time overhead of only 5%, 3× less than the state-of-the-art, while also reducing memory usage by over 2×.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '20, June 15–20, 2020, London, UK

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386030

Keywords: Code Size Reduction, Function Merging, LTO.

ACM Reference Format:
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 International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3386030

1 Introduction

The embedded system market is rapidly growing and branching out, from cars and airplanes to autonomous robots and smart cities. Embedded systems need to perform increasingly complex jobs with the support of large libraries and deep software stacks that must run on inexpensive and resource-constrained devices. These two aims are conflicting, particularly so for permanent storage and memory which already represent a significant chunk of the system area and cost.

Despite the importance of keeping code size small, compilers still make little effort to reduce it. Even when optimizing for size, their efforts are limited to disabling performance optimizations which increase size, using more compact instructions sets, and basic redundancy elimination [3, 6]. Because of that, to avoid the expensive costs of extra storage and memory, the developers of these embedded systems have to manually find ways to shrink their code, which is also costly and undesirable.

One optimization that can potentially reduce code size is function merging. Its task is to find similarities in functions and replace them with a single function that combines the functionality of the original functions while eliminating redundant code. At a high level, the way this works is that code specific to only one input function is added to the merged function but made conditional to a function identifier, while code found in both input functions is added only once and executed regardless of the function identifier.

Prior function merging methods were limited to identical or isomorphic functions, but a recent work has generalized function merging to any arbitrary pair of functions. The state-of-the-art (FMSA) [28] first represents functions as nothing more than linear sequences of instructions and labels. Then it applies a sequence alignment algorithm, developed for bioinformatics, to discover the optimal way to create pairs

of mergeable instructions from the two input sequences. Finally, it generates the merged function where aligned pairs of matching instructions are merged to a single instruction, while non-matching instructions are simply copied into the merged function.

While representing a leap forward, experiments show that FMSA fails to reduce code size in some cases where it would be intuitively expected to work. Even when handling similar functions that should be profitably merged, this algorithm may fail spectacularly, producing a merged function *larger* than the combined input functions.

Closer inspection reveals that the problem stems from the inability of this approach to handle *phi-nodes*. In SSA, *phi-nodes* merge the assignments of a single variable that arrive from different control flow paths. As such, they are closely tied to how control and data flow across basic blocks and cannot be merged without examining their control flow context. FMSA generates code directly from the aligned sequences, where control flow information has been lost, merging instructions blindly with little to no consideration for their context, so it cannot handle *phi-nodes*. It overcomes this hurdle by applying register demotion, which replaces *phi-nodes* with stack variables. This works but only by artificially increasing the size of the input functions, often by twice or more their original size, the exact opposite of what function merging tries to achieve. A final post-merging step of register promotion is supposed to reverse this code bloating but it often fails, leading to unprofitable merged functions.

Our idea is to keep the one thing that works well in FMSA, the idea of using sequence alignment on functions, and build around it a new function merging methodology that can handle directly control and data flow with no need for register demotion. Our proposed approach, SalSSA, achieves this with a new code generator for aligned functions. Instead of translating the alignment directly into a merged function, our approach generates code from the input control-flow graphs, using the alignment only to specify pairs of matching labels and instructions. The generator then produces code top-down, starting with the control flow graph of the merged function, then populating with instructions, arguments and labels, and finally with *phi-nodes* which maintain the correct flow of data. SalSSA is carefully designed to produce correct but, still, succinct code. A final post-generator stage applies a novel optimization, *phi-node coalescing*, that eliminates superfluous phi-nodes and select instructions, reducing even further the code size.

SalSSA produces functions much smaller than those produced by FMSA. In many cases, it produces profitable merged functions where FMSA fails. On average, it reduces about twice as much code as their approach, 11.4% to 14.5% compared to 5.6% to 6.2% depending on the function merging configuration. On top of that, the compile-time overhead is much lower. Sequence alignment has a quadratic relationship with function size, while the overhead of code generation and later optimization passes is proportional to function size. By avoiding register demotion, we keep input function sequences smaller and we produce smaller functions, leading to an average compilation overhead of 5%, 3× less than FMSA, and an overhead in no case more than 55%, compared to the maximum overhead of 314% for FMSA. Similarly, SalSSA uses half the amount of memory required on average by FMSA during compilation.

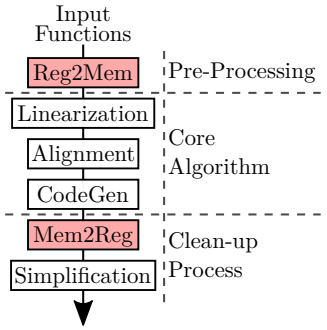


Figure 1. The sequence of operations applied by the state-of-the-art function merging. The core algorithm is composed only of Linearization, Alignment, and CodeGen, but register demotion (Reg2Mem) is necessary as a preprocessing step because CodeGen cannot handle *phi-nodes*. Register promotion (Mem2Reg) and Simplification are not required but improve the quality of the generated code. Our work introduces a more powerful CodeGen component that removes the need for register demotion.

With this paper, we make the following contributions:

- The first approach that fully supports the SSA form when merging functions through sequence alignment.
- A novel optimization called phi-node coalescing that reduces the number of phi-nodes and selections in merged functions.
- SalSSA achieves about twice as much code size reduction than the state of the art with significantly lower compilation time overheads.

2 Background

Figure 1 depicts the workflow of FMSA [28]. The state-of-the-art function merging is capable of merging any pair of functions. Its core merging algorithm first transforms each function into a linear sequence of labels and instructions. Then a sequence alignment algorithm searches for the optimal way to align two input sequences based on their matching subsequences, effectively identifying the mergeable code. The final step uses the resulting aligned sequence to directly generate code. Matching subsequences are merged into a single copy, avoiding redundancy. Non-matching segments are copied as-is into the merged function but have their execution conditioned by a function identifier.

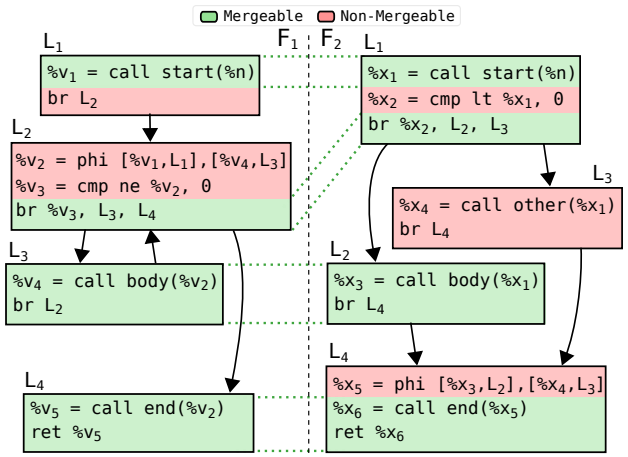


Figure 2. Original input functions to be merged, before register demotion. These simplified functions highlight a problem commonly seen in real programs.

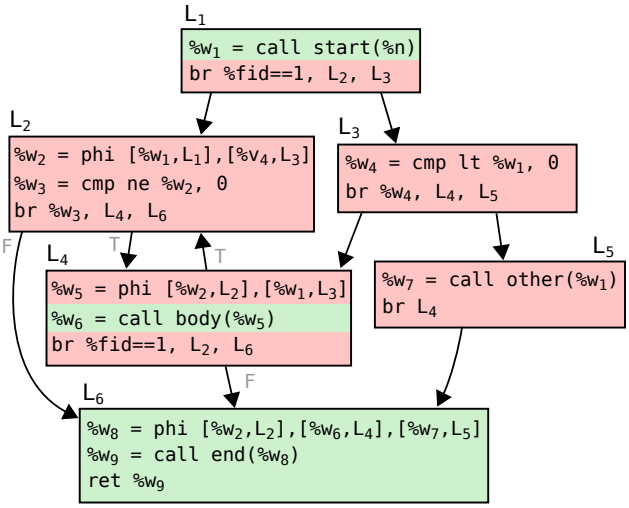


Figure 3. Desired merged function that can be produced by an expert. An extra argument called `%fid` is used to select between the two functions. This represents a gain of about 20% in the total number of instructions.

While this is all that is needed to do function merging in theory, FMSA adds an important extra preprocessing step: register demotion. Because its code generator cannot handle *phi-nodes*, it needs to apply register demotion and replace *phi-nodes* with memory operations in the stack. After code generation, register promotion is performed to transform stack operations back into *phi-nodes*, when this is possible.

The reason for FMSA to apply register demotion is clear - while *phi-nodes* are strongly coupled with the control flow, memory operations are much easier to handle as the code can be generated directly from the aligned sequences. However, as we will show in the next section, register demotion has a negative impact on both the quality of merged code and the overhead of function merging.

3 Motivating Example

As a motivation example, consider the pair of input functions shown in Figure 2. While they are artificial, they highlight and isolate a problem that frequently appears in real programs, as we discuss later in Figure 5. These two functions have enough similarity to be profitably merged. A human expert could even replace them with the function shown in Figure 3, reducing the number of instructions by about 20%.

However, before aligning and automatically merging them, FMSA has to apply register demotion, as shown in Figure 4. *Phi-nodes* are removed and memory operations are created to propagate values across basic block boundaries. The sequence alignment algorithm then identifies the matching pairs of instructions (connected green marks), keeping the rest unaligned (in red).

The problem arises when merging some of the generated memory operations. To reverse the effect of register demotion, FMSA applies register promotion on the merged code, replacing the memory operations back with *phi-nodes*. This is mandatory in FMSA in order for merged functions to be profitable, given that register demotions artificially increases the size of the functions being merged. However, in order to be promotable, a stack location must be always used directly as the immediate argument of the operations that access the location. Unfortunately, merging these instructions tend to prohibit register promotion, which results in unprofitable merge operations.

In our example, we see in Figure 4 that some of the mergeable memory operations use different locations. One such case is the highlighted pair of *store* instructions. To maintain the semantics of the two functions after merging, the target address of the merged *store* will have to be selected based on the function identifier, either $\texttt{addr}_2$ or $\texttt{addr}_3$. Because the merged *store* instruction will not use the stack address directly, but instead a selected address, this prevents register promotion from eliminating these memory operations.

This failure to remove temporarily inserted stack operations has knock-on effects beyond the few extra instructions left in the merged code. The additional memory accesses and the select statements controlling their target locations prohibit parts of the post-merge cleanup and later optimization passes. In our example, while the two original input functions had nine and ten instructions each, the merged function ends up with a total of 50 instructions, significantly larger than the two input functions put together.

This kind of undesired scenario is likely to happen when merging two distinct functions after register demotion simply due to the sheer number of memory operations it creates. Figure 5 shows the average normalized size, before and after register demotion, across all functions in each program from the SPEC CPU2006 benchmark suite. Size refers to the number of LLVM IR instructions. On average, register demotion

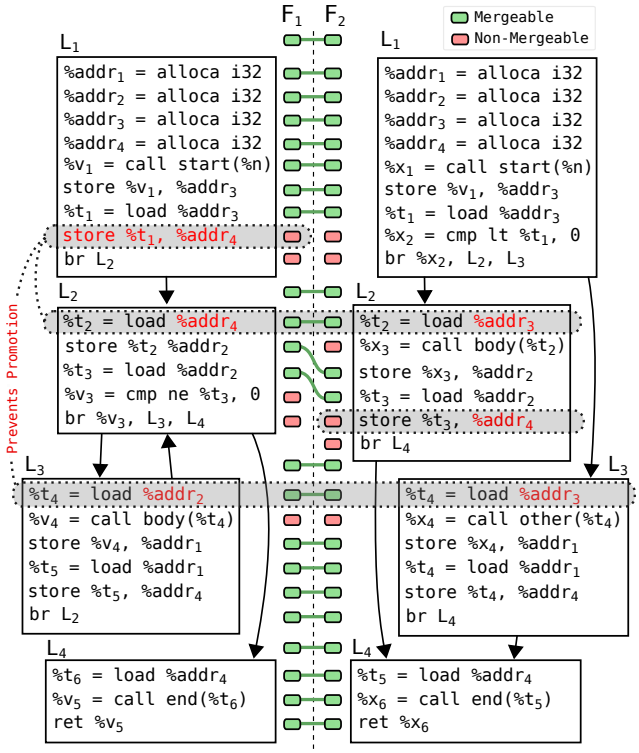


Figure 4. Aligned example functions after register demotion. The functions double in size after demotion, slowing down alignment. Merging some of the generated stack accesses will prevent eliminating them later through register promotion.

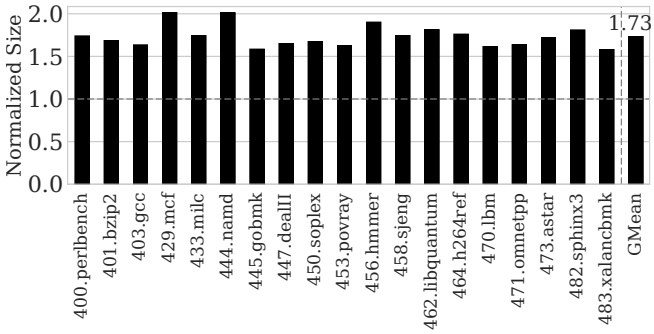


Figure 5. Average normalized function size, before and after register demotion, across all functions in each program from the SPEC 2006 benchmark suite. Register demotion increases function size by almost 75% on average.

increases function size by almost 75%, often by twice or more their original size. Even if FMSA fails to eliminate only a small portion of these extra instructions, the negative impact on the profitability of merging will be significant.

Even for cases where the merge operation is profitable, register demotion remains a problem. Demotion artificially lengthens the functions to be aligned which in turns exacerbates the compile-time overheads associated with function merging. In our example, the combined size of the two input functions more than doubles, from 14 instructions in Figure 2 to 29 instructions in Figure 4. This increase is in line with what we have seen in SPEC CPU2006, including functions with many thousands of instructions. Regardless of whether register promotion will eventually remove the extra instructions or not, the alignment algorithm itself will have to process sequences twice as long. Since the memory usage and running time of the algorithm is quadratic in the sequence length, register demotion slows it down approximately by a factor of four. For applications with large functions after register demotion, the compile-time and memory usage overheads become prohibitive.

This shows that a new solution is needed to effectively merge functions in the SSA form. Register demotion makes function merging less profitable, even stopping similar functions from merging altogether, and often leads to undesirable compilation overheads. In the rest of the paper, we show that register demotion is not required for function merging and that we can directly handle *phi-nodes*, leading to more profitably merged functions.

4 Our Approach

Properly handling *phi-nodes* requires a radical redesign in the code generator. The existing code generator produces code directly from the aligned sequence, with each instruction pair treated almost in isolation without considering any control flow context. Merging *phi-nodes* cannot work with this approach because *phi-nodes* are only understood in their control flow context.

Road map. In the rest of this section, we describe SalSSA, our novel approach for merging functions through sequence alignment with full support for the SSA form. By removing the need for preprocessing the input functions and performing register demoting, our approach is able to merge functions better and faster. Instead of translating the aligned functions directly to merged code, the SalSSA follows a top-down approach centered on the CFGs of the input functions. It iterates over the input CFGs, constructing the CFG of the merged function, interweaving matching and non-matching instructions (Section 4.1). Afterwards, all edges and operands are resolved, including appropriately assigning the incoming values to all *phi-nodes* (Section 4.2). SalSSA is designed to preserve all properties of SSA form via the standard SSA construction algorithm (Sections 4.3). Finally, SalSSA integrates a novel optimization with the SSA construction algorithm, called *phi-node coalescing*, producing even smaller merged functions (Section 4.4).

Working examples. Figure 6 shows how the functions from our motivating example align without register demotion. Here, *phi-nodes* are not aligned, similarly to how FMSA

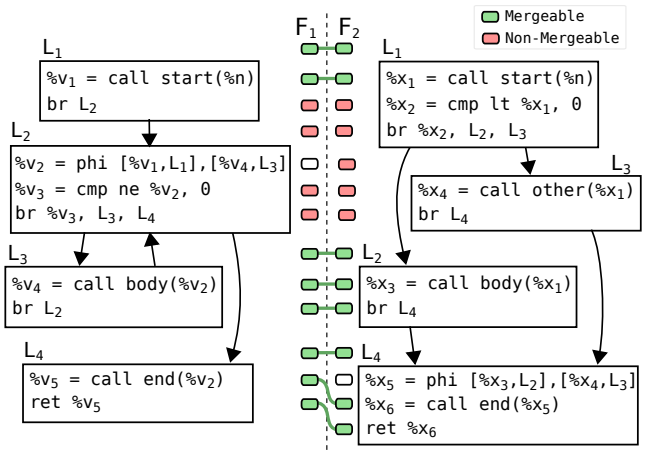


Figure 6. Example functions aligned without register demotion. *Phi-nodes* are excluded from alignment.

handles *landing-pad* instructions. We will use these as working examples to describe step by step how our new code generator works in the next subsections.

4.1 Control-Flow Graph Generation

Our code generator starts by producing all the basic blocks of the merged function. Each original block is broken into smaller ones so that matching code is separated from non-matching code and matching instructions and labels are placed into their own basic blocks. Having one block per matching instruction or label makes it easier to handle control flow and preserve the ordering of instructions from the original functions by chaining these basic blocks as needed.

Blocks with instructions that come originally from the same basic block (of either input function) are chained in their original order with branches. We use either unconditional branches or conditional branches on the function identifier depending on whether control flow out of this code is different for the two input functions. Because we have one basic block per pair of matching instructions/labels, this tends to generate some artificial branches, most of them are unconditional, but can be simplified in later stages.

Figure 7 shows the generated CFG. At this point, the only instructions that actually have their operands assigned are the branches inserted to chain instructions originating from the same input basic block. These branches have no corresponding instruction in the input functions. All other operands and edges, depicted in blue in Figure 7, will be resolved later, during operand assignment.

4.1.1 Phi-Node Generation. Our code generator treats *phi-nodes* differently from other instructions. For all alignment and code generation purposes, SalSSA treats *phi-nodes* as attached to their basic block's label; that is, they are aligned with their labels and are copied to the merged function with their labels. So, when creating a basic block for a

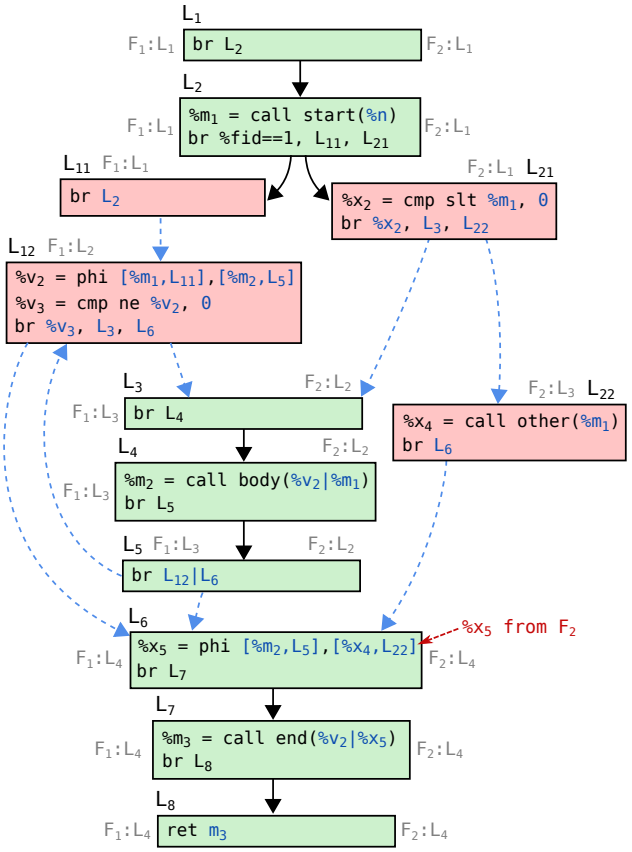


Figure 7. Merged CFG produced by SalSSA. Code corresponding to a single input basic block may be transformed into a chain of blocks, separating matching and non-matching code. The generator inserts conditional and unconditional branches to maintain the same order of instructions from the input basic block. Operands and edges highlighted in blue will be resolved by the operand assignment described in Section 4.2.

label, we also generate the *phi-nodes* associated with it. For a pair of matching labels, we copy all *phi-nodes* associated with both labels. We have decided for this approach where phi-nodes are tied to labels because phi-nodes describe primarily how data flows into its corresponding basic block. Figure 7 shows an example where *phi-nodes* are present in basic blocks with both matching or non-matching labels. The phi-node x_5 is simply copied into the merged basic block labeled L_6.

Unlike other instructions, we do not merge *phi-nodes* through sequence alignment. Instead, identical *phi-nodes* are merged during the simplification process using existing optimizations from LLVM.

4.1.2 Value Tracking. While generating the basic blocks and instructions for the merged function, SalSSA keeps track of two mappings that will be needed during operand assignment. The first one, called *value mapping*, is responsible for

```
%m2 = call body(%v2|%m1)
            |||
%s = select %fid==1, %v2, %m1
%m2 = call body(%s)
```

Figure 8. Operand selection for the `call` instruction in L_4 from Figure 7. Mismatching operands chosen with a `select` instruction on the function identifier.

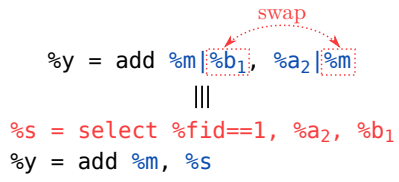


Figure 9. Optimizing operand assignment for commutative instructions. Example of a merged `add` instruction that can have its operands reordered to allow merging the two uses of `%m`, avoiding a *select* instruction.

mapping labels and instructions from the input functions into their corresponding ones in the merged function. This is essential for correctly mapping the operand values. The second one, called *block mapping*, is a mapping of the basic blocks in the opposite direction, as shown by the light gray labels in Figure 7. It maps basic blocks in the merged function to a basic block in each input functions, whenever there is a corresponding one. This *block mapping* will be needed to map control flow when assigning the incoming values of *phi-nodes* (see Section 4.2.3).

4.2 Operand Assignment

Once all instructions and basic blocks have been created, we perform operand assignment in two phases. First, we assign all label operands, essentially resolving the remaining edges in the control flow graph (dashed blue edges in Figure 7). With the control flow graph complete, we can then create a dominator tree to help us assign the remaining operands while also properly handling instruction domination.

Whenever the corresponding operands of merged instructions are different, we need a way to select the correct operand based on the function identifier. Section 4.2.1 describes how we perform label selection. In all other cases, we simply use a *select* instruction, as shown in Figure 8.

When assigning operands to commutative instructions, we also perform operand reordering to maximize the number of matching operands and reduce the need for *select* instructions. Figure 9 shows an example of a commutative instruction where an operand selection can be avoided by reordering operands. This property of commutative operations has been exploited before by other optimizations [26–28].

4.2.1 Label Selection. In LLVM, labels are used exclusively to represent control flow. More specifically, label operands

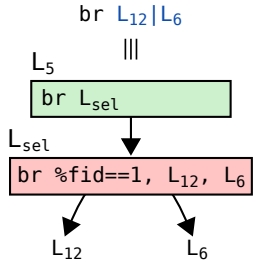
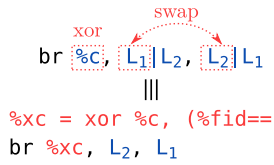


Figure 10. Label selection for mismatched terminator instruction operands L_{f1} and L_{f2} corresponding to labels of two different basic blocks. We handle control flow in a new basic block, L_{sel} with a conditional branch on the function identifier targeting the two labels. We use the label of the new block as the merged terminator operand.



(a) Rule for conditional branches with swapped label operands.

%c	%fid==1	%xc
0	0	0
0	1	1
1	0	1
1	1	0

(b) The truth table of the xor operation.

Figure 11. Optimizing label assignment for conditional branches. Example of a merged `br` instruction that can have its label operands reordered, trading two label selections by one xor operation.

are used by terminator instructions, where they specify the destination basic block of a control flow transfer, or to represent incoming control flow in a *phi-node* instruction.

Whenever assigning the operands of a merged terminator instruction, if there is a label mismatch between the two input functions, we need a way to select between the two labels depending on the executed function. We do so by creating a new basic block with a conditional branch on the function identifier to each one of the mapped labels. Then we use the new block's label as the operand of the merged terminator instruction. Figure 10 illustrates a CFG that handles label selection for a merged terminator instruction.

Figure 11 shows a special case where we can also perform operand reordering on conditional branches that follow a specific pattern. When merging two conditional branches with matching label operands, except for their order, instead of creating two label selections, we can simply apply an *xor* operation on the condition and the function identifier, swaping the label operands for the true-value of the function identifier. As shown in Figure 11b, the xor operation flips the value of the condition for the true-value of the function identifier, preserving the semantic of the conditional branch. This optimization adds the cost of one xor operation to avoid the cost of two label selections, which are implemented with branch instructions as shown in Figure 10.

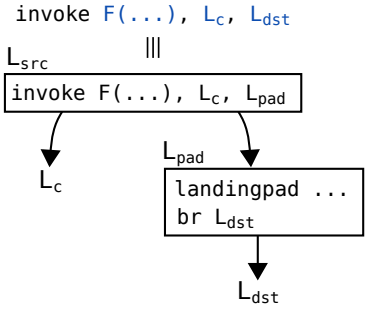


Figure 12. Landing blocks are added after operand assignment and are assigned to invoke instructions as operands.

4.2.2 Landing Blocks.

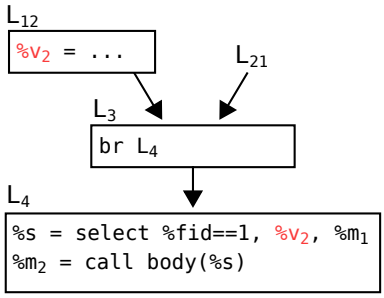
Most modern compilers, including GCC and LLVM, implement the zero-cost Itanium ABI for exception handling [11], which is known as the *landing-pad* model. This model has two main components: (1) invoke instructions that have two successors, one that continues when the call succeeds as per normal, and another, usually called the *landing pad*, in case the call raises an exception, either by a throw or the unwinding of a throw; (2) landing-pad instructions that encode which action is taken when an exception is raised. A landing pad must be the immediate successor of an invoke instruction in its unwinding path. The code generator must ensure that this model is preserved.

Our new code generator delays the creation of landing-pad instruction until the phase of operand assignment. Once we have concluded the remapping of all label operands of an invoke instruction, regardless of whether they are merged or non-merged code, we create an intermediate basic block with the appropriate landing-pad instruction. Then we assign the label of this landing block as the operand of the invoke instruction, as shown in Figure 12.

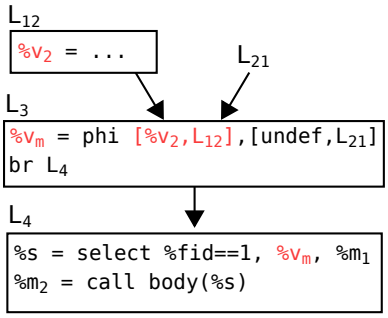
4.2.3 Phi-Node's Incoming Values.

There are two distinct cases for *phi-nodes*: being associated with a matching or with a non-matching label. In both cases, *phi-nodes* are only copied from their input functions and they are not merged. So each *phi-node* in the merged function should capture the incoming flows present in the corresponding *phi-node* of their input function. For matching labels, each *phi-node* in the merged function will have additional incoming flows specific to the *other* input function but these flows should have undefined values.

To assign the incoming values of a *phi-node*, SalSSA iterates over all predecessors of its parent basic block and uses the *block mapping* to discover each predecessor's corresponding basic block in the input function. If such a basic block is found, then SalSSA obtains the incoming value associated with that predecessor from the *value mapping*. Otherwise, an undefined value, which by construction should never be actually used, is associated with that predecessor.



(a) Example where the dominance property is violated.



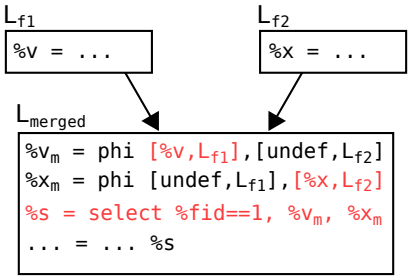
(b) The dominance property is restored by placing phi-nodes where needed.

Figure 13. Example of how SalSSA uses the standard SSA construction algorithm to guarantee the dominance property of the SSA form.

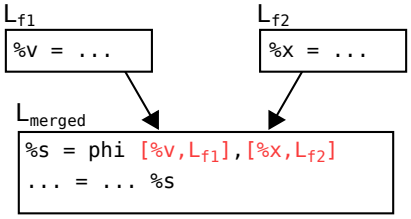
4.3 Preserving the Dominance Property

The code transformation process described so far could violate the *dominance property* of the SSA form. This property states that each use of a value must be dominated by its definition. For example, an instruction (or basic block) dominates another if and only if every path from the entry of the function to the latter goes through the former. Figure 13a gives one example extracted from Figure 7 where the dominance property is violated during code transformation. For this example, the dominance property is violated because $\%v_2$ is defined in block L_{12} and used in block L_4, but the former does not dominate the latter since there is a alternative path through L_{21}.

SalSSA is designed to preserve the dominance property to conform with the SSA form. It achieves this using a two-step approach. It first adds a *pseudo-definition* at the entry block of the function where names are defined and initialized with an *undefined* value. This guarantees that every register name will be defined on basic blocks from both functions. Then, SalSSA applies the standard SSA construction algorithm [9, 10], which guarantees both the dominance and the single-reaching definition properties of the SSA form. We note that our implementation uses the standard SSA construction algorithm provided by LLVM for register promotion. This algorithm guarantees that names have a single definition by placing extra phi-nodes where needed so that instructions



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 14. Phi-node coalescing reduces the number of phi-nodes and selections.

can be renamed appropriately. Figure 13b shows how the property violation in Figure 13a can be corrected using this strategy.

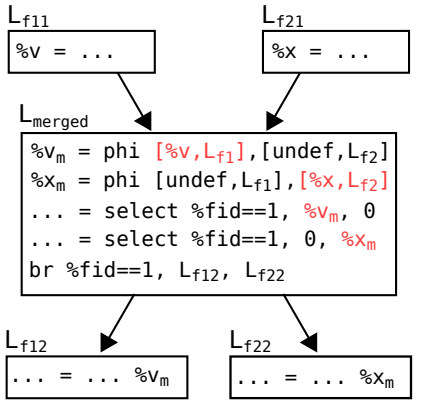
4.4 Phi-Node Coalescing

The approach described in Section 4.3 guarantees the correctness of the SSA form but generates extra phi-nodes and registers which increase register pressure and might lead to more *spill code*. In this section, we describe a novel optimization technique, *phi-node coalescing*, that SalSSA uses to lower register pressure.

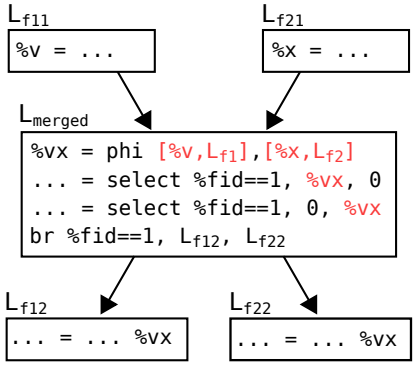
Figure 14 illustrates such an optimization opportunity. SalSSA is merging an instruction with different arguments, so it needs to select the right one based on the function identifier. The two arguments though, `v` and `x`, have *disjoint definitions*, i.e. they have non-merged definitions from different input functions. Using the standard SSA construction algorithm would result in the sub-optimal code shown in Figure 14a. This code inserts two trivial phi-nodes to select, again, `v` or `x` based on the executed function. SalSSA optimizes this code by coalescing both phi-nodes into a single one and removing the selection statement. As shown in Figure 14b, the optimized version has a smaller number of instructions and phi-nodes.

This transformation is valid because a value definition that is exclusive to a function will never be used when executing the other function. Figure 15 shows another example illustrating that even disjoint definitions that have no user instructions in common can be coalesced, reducing the number of phi-nodes.

Since SalSSA is aware of which basic blocks are exclusive to each function, it can choose a pair of disjoint definitions



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 15. Reducing the number of phi-nodes by coalescing disjoint definitions with no user instructions in common.

for coalescing. Given a pair of disjoint definitions, SalSSA assigns the same name for both of them before applying the SSA reconstruction. SalSSA coalesces the set of definitions that violate the dominance property. Two definitions can be paired for coalescing if they are disjoint and have the same type. The optimization pairs disjoint definitions that maximize their live range overlap since the goal is to avoid having register names live longer than they should, reducing register pressure.

Formally, the heuristic implemented in our phi-node coalescing can be described as follows. Given a set $S_1 \times S_2$ of disjoint definitions that violate the dominance property, the optimization chooses pairs $(d_1, d_2) \in S_1 \times S_2$ that maximize the intersection $UB(d_1) \cap UB(d_2)$, where $UB(d)$ is the set $\{Block(u) : u \in Users(d)\}$.

Phi-node coalescing allows SalSSA to produce smaller merged functions and reduce code size. Consequently, it also enables more functions to be profitably merged.

5 Evaluation

In this section, we compare SalSSA against the state-of-the-art algorithm of function merging by sequence alignment, FMSA [28]. We first present the code size reduction on the

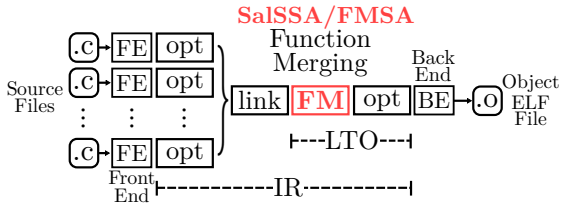


Figure 16. Compilation pipeline used for the evaluation. Both SalSSA and FMSA are applied in LTO mode.

final object file. We then evaluate the compilation overhead and impact on program performance.

5.1 Experimental Setup

Most of our experiments directly compare SalSSA against FMSA [28]. We use the same compilation pipeline as our prior work [28], depicted in Figure 16. Both function merging optimizations are implemented in LLVM version 11.

Our approach uses the same fingerprint-based ranking mechanism as FMSA to decide which functions to attempt to merge. This strategy uses a configurable *exploration threshold*, t, to control how many different functions to attempt to merge with each function before selecting the most profitable merge or give up. A larger exploration threshold (t) is likely to lead to better code size reduction, but comes at the cost of longer compile time. Like FMSA, we also use three different exploration thresholds where $t = \{1, 5, 10\}$.

We evaluated SalSSA and FMSA on all C/C++ benchmarks of the SPEC CPU benchmark suite [30], both the 2006 and 2017 versions, targeting the Intel x86 architecture, and on the MiBench embedded benchmark suite targeting the ARM Thumb architecture. We run all experiments on a dedicated server with a quad-core Intel Xeon CPU E5-2650, 64 GiB of RAM, running Ubuntu 18.04.3 LTS. To minimize the effect of measurement noise, compilation and runtime overhead experiments were repeated 5 times.

5.2 Evaluation on SPEC CPU

Figures 17 reports the code size reduction on linked objects over the LLVM link-time optimizer (LTO). SalSSA significantly improves FMSA. With the lowest exploration threshold, SalSSA on average reduces the compiled code size by 9.3% and 7.9% on SPEC CPU2006 and CPU2017 respectively. These translate to nearly twice or above as much as FMSA, which achieves a 3.8% and 4.1% reduction on SPEC CPU2006 and CPU2017 respectively. The highest reductions are seen for `447.dealII` and `510.parset_r`, over 40% reduction. They are mainly due to the heavy use of template functions which leads to multiple similar functions. Other C++ programs display similar behavior, where SalSSA also achieves good code size reduction. SalSSA also gives remarkable code size reduction in many C programs, such as `456.hmmer`, `462.libquantum`, and `482.sphinx3`.

SalSSA outperforms FMSA for multiple benchmarks. The more pronounced cases are for `444.namd`, `456.hmmer`, `462.libquantum`, `447.dealII`, and `482.sphinx3` from SPEC CPU2006, as well as `508.namd_r`, `619.lbm_s`, `644.leela_s` and `657.xz_s` from SPEC CPU2017. These benchmarks were heavily affected by register demotion, as illustrated in Figure 5 for SPEC CPU2006. Similar to our motivating example in Section 3, when two non-identical functions have stack operations for nearly half of their instructions, misalignments become likely; these misalignments prohibit eliminating the merged stack operation through register promotion. This issue reduces the profit gained by FMSA. In some cases, like `619.lbm_s` and `625.x264_s`, the profitability cost model can fail, resulting in sufficient false positives to cause code bloating. We will discuss this further in the next section.

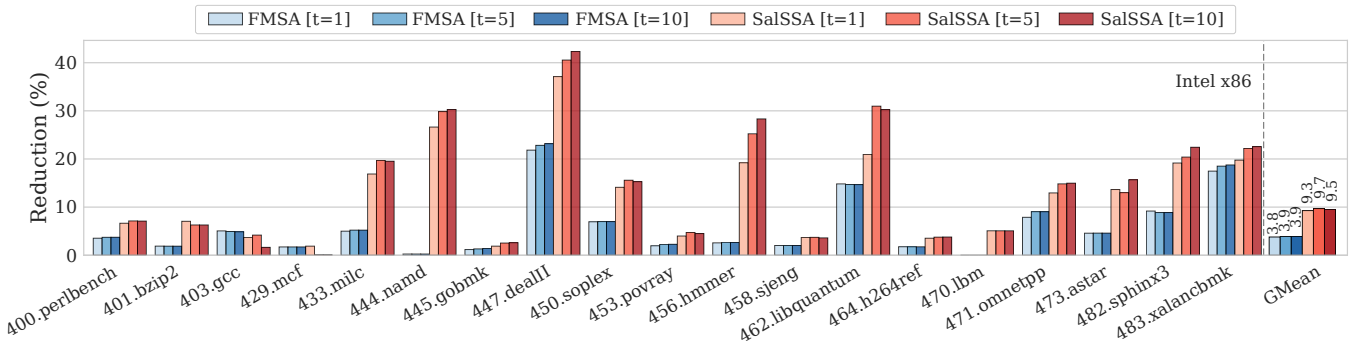
5.3 Evaluation on MiBench

To evaluate the effectiveness of SalSSA on embedded systems, we apply it to the MiBench embedded benchmark suite on the ARM Thumb architecture. We note that by having function merging implemented at the IR level, our approach can be equally applied to any target architecture supported by the compiler.

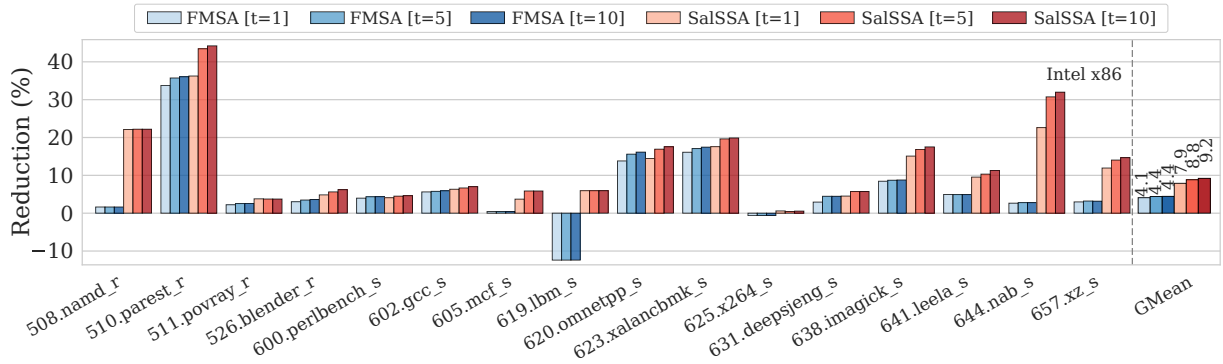
The MiBench suite is a collection of short C programs, each one composed of a small number of functions. When optimizing programs with a small number of functions, function merging optimizations will have fewer opportunities to find pairs of profitably merged functions. For example, the `qsort` program in MiBench has only two functions; as a result, neither FMSA nor SalSSA is able to merge them. As shown in Table 1, the same happens for other programs in the MiBench suite.

Figure 18 shows that SalSSA improves significantly over FMSA, achieving a geo-mean reduction of 1.4% to 1.6%, about twice as much as FMSA. This improvement comes from SalSSA's capability of generating better-merged functions, which leads to a larger number of profitable merge operations, as confirmed by Table 1.

Because FMSA requires register demotion to be applied to all functions before it can even attempt to merge them, FMSA ends up changing all functions even if no profitable merge operation is found. Figure 18 shows the effect of this preprocessing phase (denoted as *FMSA Residue*), which is obtained by running FMSA but not committing any merge operation. This FMSA Residue is the reason why FMSA sometimes has a non-zero code-size reduction (e.g., `adpcm_c`, `FFT`, `patricia`) despite not merging any functions. Since FMSA Residue might have an impact on the heuristics of later optimizations and code generation, its impact is almost random, sometimes being positive or negative on code-size. The impact of FMSA Residue is more noticeable in small programs, such as those found in MiBench, while in SPEC2006 it increases code size by only 0.02%, on average. To fix the



(a) Results on SPEC CPU 2006.



(b) Results on SPEC CPU 2017.

Figure 17. Linked object size reduction over LLVM LTO when performing function merging with SalSSA or FMSA on SPEC CPU 2006 (a) and 2007 (b). Each approach was evaluated using three different exploration thresholds. On SPEC CPU2006, SalSSA reduces code size by 9.3% to 9.7% on average, almost twice as much as FMSA. On SPEC CPU2017, SalSSA reduces code size by 7.9% to 9.2% on average, more than twice as much as FMSA.

issue highlighted by FMSA Residue, we would need to add an extra bookkeeping step of cloning all original functions so that we can rollback if they are not profitably merged. Fixing that would only increase unnecessarily the optimization complexity, but SalSSA offers a better solution where only merged functions are affected.

An interesting case is observed with both `cjpeg` and `djpeg`. Although SalSSA, with exploration threshold $t = 1$, increases code size, it is merging a superset of the pairs of functions merged by FMSA with $t = 1$. If we limit SalSSA to merge exactly the same pairs merged by FMSA, it ends up with about the same or slightly better results than FMSA. This suggests that the marginal code-size increase observed with SalSSA is a result of false positives from the profitability cost model, i.e., it allows unprofitable merge operations to be committed. Since `cjpeg` and `djpeg` share most of their code base, we can indeed confirm that a subset of the pairs of functions merged by SalSSA, for both benchmarks, should have been classified as unprofitable as merging them increases the code size. However, with higher exploration thresholds, namely, $t = 5$ and $t = 10$, SalSSA surpasses FMSA in code-size reduction, although it still includes all pairs merged with the exploration threshold $t = 1$.

Figure 19 shows a breakdown for each merge operation performed by SalSSA, with exploration threshold $t = 1$, on the `djpeg` benchmark. We measured the impact of each merge operation, in isolation, to the size of the final object file. Although each one of these merge operations have a very small contribution to the final code size, the profitability cost model failed enough to result in an overall code increase of about 0.3%.

Both SalSSA and FMSA use the same profitability cost model. The limitations observed on `cjpeg` and `djpeg` also appear in SPEC2017 with FMSA. This stems from the fact that several transformations will still be applied to the code during late optimizations and the back end, and these changes are not captured by the profitability cost model.

5.4 Further Analysis

We also provide a breakdown showing the impact of phi-node coalescing on code size. Figure 20 shows the impact of our phi-node coalescing optimization technique (see Section 4.4). This diagram compares SalSSA to a variant without phi-node coalescing (SalSSA-NoPC) and FMSA. On average, this technique gives an additional 1.2% on code size reduction.

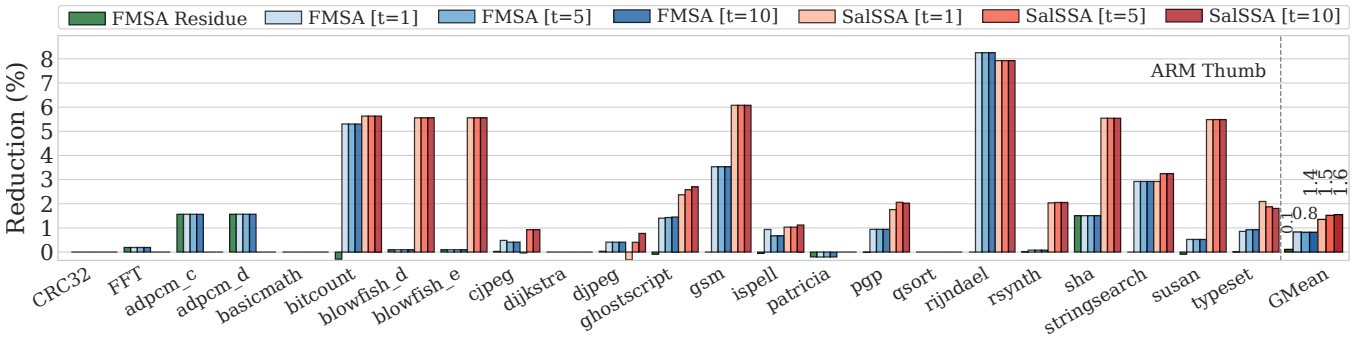


Figure 18. The percentual reduction in size of the linked object files, targeting the ARM architecture. We evaluate SalSSA or FMSA over the LLVM LTO on the MiBench embedded benchmark suite. Each approach was evaluated using three different exploration thresholds. SalSSA achieves a geo-mean reduction of 1.4% to 1.6%, about twice as much as FMSA.

Table 1. Number and size of functions present in each MiBench benchmark just before function merging, as well as number of merge operations applied by each technique.

Benchmarks	#Fns	Min/Avg/Max Size	FMSA[t=1]	SalSSA[t=1]
CRC32	4	8/23.75/37	0	0
FFT	7	6/45.43/131	0	0
adpcm_c	3	35/68.33/93	0	0
adpcm_d	3	35/68.33/93	0	0
basicmath	5	4/60/204	0	0
bitcount	19	4/20.58/56	3	3
blowfish_d	8	1/231.38/790	0	1
blowfish_e	8	1/231.38/790	0	1
cjpeg	322	1/92.76/1198	7	26
dijkstra	6	2/31.5/83	0	0
djpeg	310	1/91.31/1198	10	28
ghostscript	3452	1/50.36/3749	211	327
gsm	69	1/92.42/696	6	9
ispell	84	1/97.08/1004	3	8
patricia	5	1/73.6/160	0	0
pgp	310	1/80.39/1706	8	19
qsort	2	11/45.5/80	0	0
rijndael	7	45/444.14/1182	1	1
rsynth	47	1/83.89/716	1	2
sha	7	12/49.71/147	0	1
stringsearch	10	3/41/81	1	1
susan	19	15/275.21/1153	1	2
typeset	362	1/327.61/11744	27	53

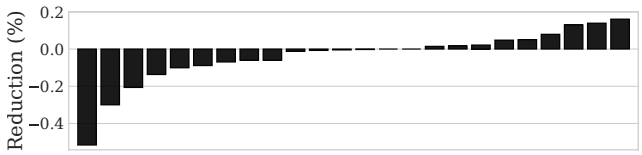


Figure 19. A breakdown of SalSSA[$t = 1$] on the `djpeg` benchmark. The actual contribution to the final code size for each merge operation deemed profitable by the cost model.

For `444.namd`, it enables an extra 7% reduction on the code size, demonstrating the great advantage of the technique.

Figure 21 provides further insight into the gains of SalSSA. The figure shows the total number of profitable merging attempts for the lowest exploration threshold. While FMSA has only 9,271 profitable merge operations, SalSSA has 12,224, an increase of 31% on the number of profitable merges. Much of the improvement we observe in code size reduction comes from producing profitable merged functions where FMSA fails to gain any profit, not just from increasing the profit.

5.5 Memory Usage

Because the sequence alignment algorithm [24, 28] (used by FMSA and SalSSA) has a quadratic space complexity over the length of the sequences, the difference in the size of the functions caused by register demotion translates directly to the differences in memory usage.

Figure 22 shows the peak memory usage across the SPEC CPU2006 suite. To isolate the impact of other compilation passes, we measure the memory usage only when running the function merging optimization. As expected, avoiding register demotion has the added benefit of lowering the memory footprint of the compilation pass. On average, SalSSA uses half the amount of memory required by FMSA. The improvements on memory usage shown in this Figure 22 directly reflects the difference shown in Figure 5.

Both FMSA and SalSSA starts merging from the largest to the smallest functions. For the `403.gcc` benchmark, the first pair of functions considered for a merging is the pair `recog_16` and `recog_26` that originally contains 20,688 and 16,043 instructions, respectively, but after register demotion grow to 36,508 and 28,899. This pair of extremely large functions is responsible for the peak in memory usage when optimizing this benchmark. FMSA uses a total of 6.5 GB of memory while SalSSA is able to reduce it down to 2.4 GB. A total of 2.7× reduction on peak memory usage. Although this is the most critical benchmark in terms of absolute numbers, a similar trend appears in most of the other benchmarks. By reducing the memory overhead of compilation, SalSSA thus can target a larger codebase over FMSA.

5.6 Compilation Time Overhead

Figure 24 shows the normalized compile-time for *the entire* compilation process on SPEC CPU2006. The min-max bar

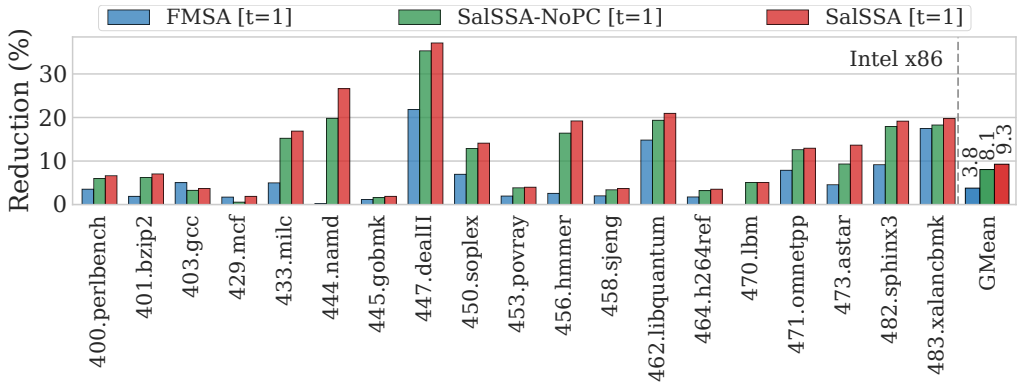


Figure 20. Evaluation of the impact of phi-node coalescing on the size of the final object file. SalSSA-NoPC, which includes phi-node coalescing, has a measurable benefit over the alternative without phi-node coalescing (SalSSA-NoPC). When enabled, phi-node coalescing achieves up to 7% of code size reduction on top of SalSSA-NoPC.

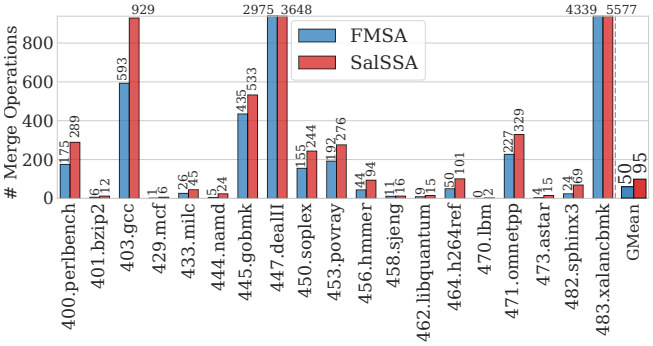


Figure 21. Total number of profitable merge attempts for SalSSA and FMSA on 19 SPEC CPU2006 benchmarks. For both cases, we used the lowest exploration threshold (t=1). SalSSA achieves 31% more profitable merge operations.

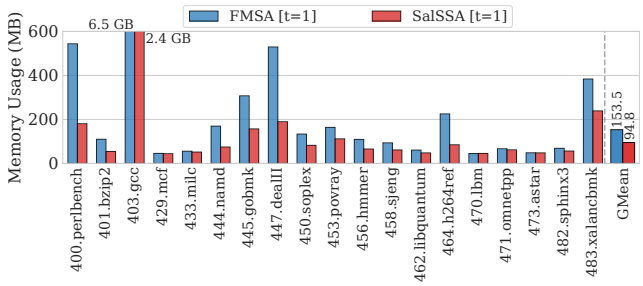


Figure 22. Peak memory usage during compilation time on the SPEC CPU2006 benchmark. On average, SalSSA requires less than half the memory used by FMSA.

in the diagram gives the 95% confidence interval across different compile-time measurements of a benchmark. SalSSA incurs modest compile-time overhead with an average 5% increase in the compile-time when using the lowest exploration threshold ($t = 1$). This represents a 3x reduction in the compile-time overhead compared to the 14% overhead from FMSA with the same exploration threshold. When using the

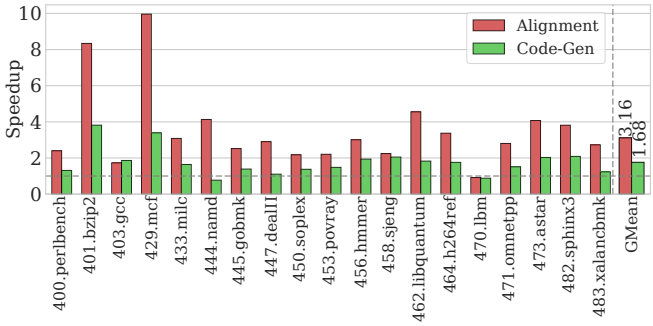


Figure 23. Speedup over the accumulated time spent on both sequence alignment and code generation. SalSSA produces significantly less overhead than the state-of-the-art FMSA.

largest exploration threshold ($t = 10$), we observe a 3.7x reduction in the compile-time overhead. The improvement is due to not only less time spent performing the optimization itself but also less work for the remaining compilation process since we reduce the size of the produced code. We also observe similar overhead improvement on SPEC CPU2017.

Figure 23 shows the speedups obtained by SalSSA for the sequence alignment and the code generator. These two stages of function merging benefit most from our techniques. As suggested earlier, both stages are accelerated because the compiler has shorter sequences to operate on under SalSSA over FMSA. The results given in Figure 23 and Figure 5 follow a very similar trend. These confirm our intuition described earlier in Section 3.

Since the sequence alignment algorithm is also quadratic in time over the length of the sequences, we get a quadratic speedup by avoiding register demotion with SalSSA. Code generation is linear on the size of the functions resulting in proportional speedups in compile-time. For a couple of cases, however, the pressure put on the clean-up phase can negate those gains.

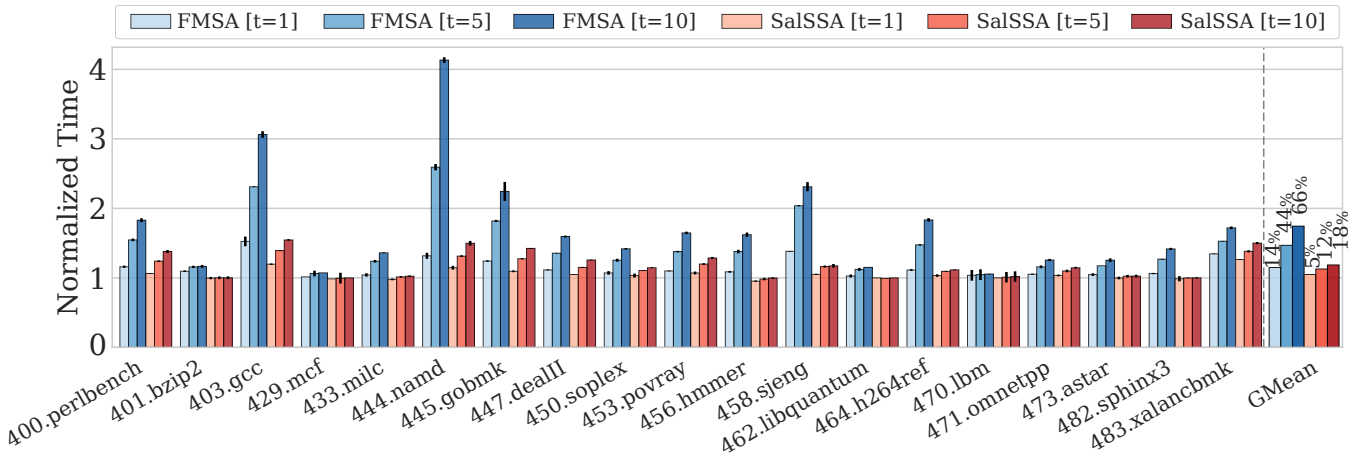


Figure 24. End-to-end compile-time for SalSSA and FMSA for three different exploration thresholds and 19 different SPEC CPU2006 benchmark. Compile-time is normalized to that of the baseline with no function merging. SalSSA reduces the overhead of function merging by 3× to 3.7× on average.

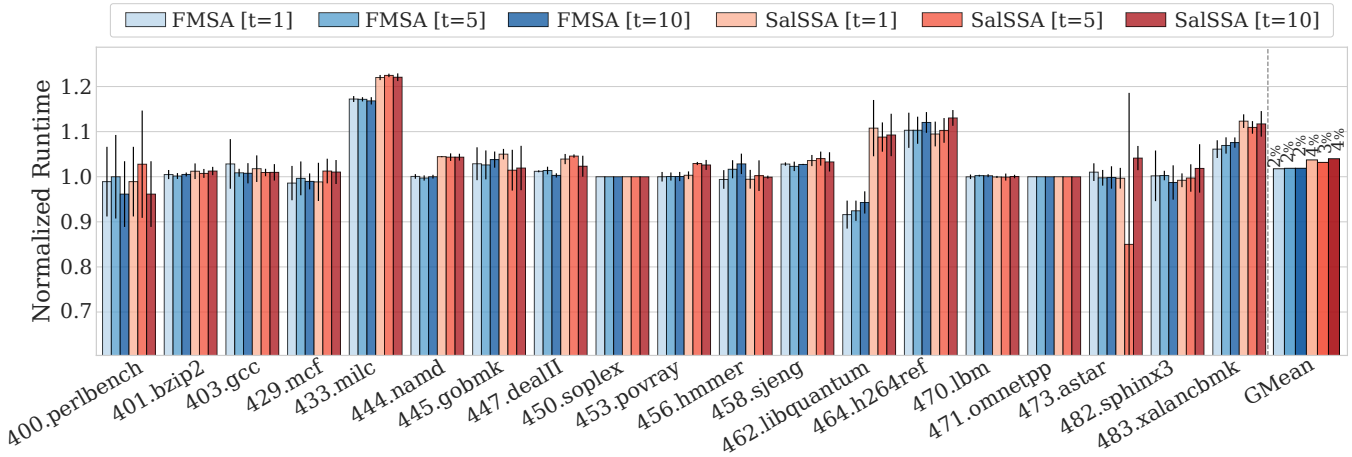


Figure 25. Comparison between the runtime impact from FMSA and SalSSA. Our approach increases the runtime overhead because it merges more functions. For most benchmarks, the overhead is small. For the rest, profiling-directed merging would eliminate the overhead.

5.7 Performance Overhead

The primary goal of function merging is to reduce code size. Nevertheless, it is also important to keep the impact on the program runtime as low as possible. Figure 25 shows the normalized execution time, where the min-max bar shows the 95% confidence interval across different runs. Overall, SalSSA has an average overhead of about 4% on programs' runtime. For most benchmarks, there is no statistically significant difference between the baseline and the optimized binary. For the rest, profiling information could be used to avoid adding overhead when mergeable code is in the most frequently executed code path.

6 Related Work

Compiler-based code size reduction is certainly not a new research topic. Prior work achieves this by either replacing the target code with a smaller, semantically-equivalent code [22, 32], or removing or combining redundant code [5–7, 12, 15, 18, 21]. Our work falls into the latter category.

6.1 Function Merging

Link-time code optimizers like [1, 19, 31] merge text-identical functions at the bit level. However, such solutions are platform-specific and need to be adapted for each object code format and hardware architecture. However, GCC and LLVM [2, 20] also provide an optimization for merging identical functions at the IR level and hence is agnostic to the target hardware. Unfortunately, they can only merge fully identical functions

with at most type mismatches that can be losslessly cast to the same format. The work presented by von Koch et al. [14] advanced this simple merging strategy by exploiting the CFG isomorphism of two functions. However, it requires two mergeable functions to have identical CFGs and function types, where the two functions can only differ between correspoding instructions, specifically, in their opcodes or the number and types of the input operands. The state-of-the-art technique, FMSA [28], lifts most of the restrictions imposed by prior techniques [2, 14, 20]. Although achieving impressive results, it does not directly handle *phi-nodes* which are fundamental to the SSA form. Instead, it applies register demotion to replace all such nodes with memory operations, in an attempt to simplify the code generation processes. As we have shown in this paper, such a strategy comes at the cost of poor merge results, larger memory footprint and longer compilation time. Our work avoids this pitfall with a new code generator capable of handling *phi-nodes* properly and completely bypassing register demotion. This leads to better code reduction performance and faster compilation time over the state-of-the-art.

Procedural abstraction [13, 21] and function merging are two different but complementary code optimization techniques. Procedural abstraction extracts identical code segments to separate functions and replaces the original code segment with a function call. Procedural abstraction typically only works on single basic blocks or single-entry, single-exit code regions, which are text identical. By contrast, function merging works on whole functions and does not necessarily require the functions to be fully identical.

In a broader context, code similarity detection is a heavily studied field. It has been used for a wide range of tasks including GPU code optimization [8], code maintenance [23, 23, 33] and software development tasks like code clone detection [29]. Unlike many of these tasks where a certain degree of approximation may be acceptable, function merging requires a precise analysis of code similarity. To this end, we use the same sequence alignment technique proposed in the state-of-the-art [28].

6.2 Phi-Node Coalescing

Some work in the literature also uses the term *"phi coalescing"* but in the context of register allocation. For example, the C2 compiler implements *"phi coalescing"* as part of its aggressive-coalescing optimization s performed during register allocation [16, 17]. In this context, *"phi coalescing"* refers to a transformation where the input values of a single phi-node are coalesced into the same register, avoiding unnecessary copies [16, 25]. While our work builds upon the past foundations of register allocation and coalescing [4, 25], *phi-node coalescing* in this paper refers to coalesce different phi-nodes into one. Our goal is to reduce the total number of phi-nodes after merging two functions. This is different from prior work that aims to use a single register to represent a phi-node after register allocation [16, 17].

7 Conclusion

We have presented SalSSA, a novel compiler-based function merging technique with full support for the SSA form. Unlike the previous state-of-the-art, which has to apply register demotion to eliminate the commonly used *phi-nodes* in SSA, SalSSA directly processes *phi-nodes* using a more powerful code generator. As a result, SalSSA avoids the code bloating problem introduced by register demotion and increases the chances of generating profitable merged functions. We have implemented SalSSA in LLVM and evaluated it on the SPEC CPU2006 and CPU2017 benchmark suites. SalSSA delivers on average 9.5% code reduction for the lowest exploration threshold. Compared to the previous function merging state-of-the-art, SalSSA achieves 2× more reduction on binary size with 3× less compile-time overhead and less than half the amount of memory required by it.

For future work, we plan to investigate the application of phi-node coalescing outside function merging. In order to avoid code size degradation, we also plan to improve the compiler's built-in static cost model for code size estimation. As a future work, we can also analyze the interaction between function merging and other optimizations such as inlining, outlining, and code splitting. Finally, we also plan to incorporate instruction reordering into function merging to maximize the number of matches between the functions regardless of the original code layout.

Acknowledgment

This work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/P003915/1 (SUMMER) and EP/M01567X/1 (SANDeRs). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] [n.d.]. Microsoft Visual Studio. Identical COMDAT Folding. https://msdn.microsoft.com/en-us/library/bxwfs976.aspx.

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

[3] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. 1997. Value Numbering. *Software: Practice and Experience* 27, 6 (1997), 701–724.

[4] Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. 2002. Fast Copy Coalescing and Live-Range Identification. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 25–32.

[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.

[6] John Cocke. 1970. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization*. ACM, New York, NY, USA, 20–24.

[7] 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.

[8] 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.

[9] 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.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490.

[11] Christophe de Dinechin. 2000. C++ Exception Handling. *IEEE Concurrency* 8, 4 (Oct. 2000), 72–79.

[12] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415.

[13] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen. 2007. Graph-Based Procedural Abstraction. In *International Symposium on Code Generation and Optimization (CGO'07)*. 259–270.

[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.

[15] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. 1997. Code Compression. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation (PLDI '97)*. ACM, New York, NY, USA, 358–365.

[16] Java OpenJDK. [n.d.]. The C2 Register Allocator. https://wiki.openjdk.java.net/display/HotSpot/The+C2+Register+Allocator. Page visited on 2020 and last modified on Apr 15, 2013.

[17] Java OpenJDK. [n.d.]. C2 Register Allocator Notes. https://wiki.openjdk.java.net/display/HotSpot/C2+Register+Allocator+Notes. Page visited on 2020 and last modified on Apr 9, 2009.

[18] 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.

[19] Doug Kwan, Jing Yu, and B. Janakiraman. 2012. Google's C/C++ toolchain for smart handheld devices. In *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*. 1–4.

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

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

[22] 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.

[23] Webb Miller and Eugene W. Myers. 1985. A file comparison program. *Software: Practice and Experience* 15, 11 (1985), 1025–1040.

[24] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443 – 453.

[25] Fernando Magno Quintão Pereira and Jens Palsberg. 2009. SSA Elimination after Register Allocation. In *Compiler Construction*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–173.

[26] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 206–216.

[27] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 163–174.

[28] 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.

[29] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168.

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

[31] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit*.

[32] 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.

[33] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.