Reducing Code Size with Function Merging

Rodrigo Caeteano de Oliveira Rocha

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2020

Abstract

Acknowledgements

TODO.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Rodrigo Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. "Function merging by sequence alignment." In IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 149-163. Best Paper Award. 2019.
- Rodrigo Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. "Effective function merging in the SSA form." In ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. - . 2020.

(*Rodrigo Caeteano de Oliveira Rocha*)

Table of Contents

Chapter 1

Introduction

1.1 Sources of Similarities

Note that functions that are identical at the IR or machine level are not necessarily identical at the source level. Figure 1.1 shows two real functions extract from the 447.dealII program in the SPEC CPU2006 [5] benchmark suite. Although these two functions are not identical at the source level, they become identical after a template specialization and some optimizations are applied, in particular, constant propagation, constant folding, and dead-code elimination. Specializing `dim` to 1 enables to completely remove the loop in the function `PolynomialSpace`. Similarly, specializing `dim` to 1 results in only the first iteration of the loop in the function `TensorProductPolynomials` being executed. The compiler is able to statically analyze and simiplify the loops in both functions, resulting in the identical functions shown at the bottom of Figure 1.1.

```
template <int dim>
unsigned int PolynomialSpace<dim>::
compute_n_pols (const unsigned int n) {
  unsigned int n_pols = n;
  for (unsigned int i=1; i<dim; ++i) {
    n_pols *= (n+i);
    n_pols /= (i+1);
  }
  return n_pols;
}
```

```
template <int dim> inline
unsigned int TensorProductPolynomials<dim>::
x_to_the_dim (const unsigned int x) {
  unsigned int y = 1;
  for (unsigned int d=0; d<dim; ++d) {
    y *= x;
  }
  return y;
}
```

After template specialization and applying optimizations:

```
unsigned int PolynomialSpace<1>::
compute_n_pols(const unsigned int n) {
  return n;
}
```

```
unsigned int TensorProductPolynomials<1>::
x_to_the_dim(const unsigned int x) {
  return x;
}
```

Figure 1.1: Two function extracted from the 447.dealII benchmark that are not identical at the source level, but after applying template specialization and optimizations they become identical at the IR level.

Chapter 2

Related Work

2.1 Identical Code Folding in Linkers

Google developed an optimisation for the *gold* linker that merges identical functions on a bit-level [3, 6]. After placing each function in a separate ELF section, they identify function sections that have their *text* section bit-identical and also their relocations point to sections that are identical. A simpler version of this optimisation was also offered by the MSVC linker [1];

2.2 Identical Function Merging

A similar optimisation for merging identical functions, but instead at the intermediate representation (IR) level, is also offered by both GCC and LLVM [2, 4]. This optimisation is only flexible enough to accommodate simple type mismatches provided they can be bitcasted in a losslessly way. Its simplicity allows for an efficient exploration approach based on computing the hash of the functions and then using a tree to identify equivalent functions based on their hash values.

We define a congruent group as a set of functions that are candidates for function equality. To create congruent groups, we build a compound hash value for each previously parsed function. This hash is cheap to compute, and has the property that if function $F = G$ according to the comparison function, then $hash(F) = hash(G)$. Therefore, as an optimisation, two functions are only compared if they have the same hash. This consistency property is critical to ensuring all possible merging opportunities are exploited. Collisions in the hash affect the speed of the pass but not the correctness or determinism of the resulting transformation.

After that, the pass sorts each function to a congruent class according to its hash value. All functions in the module, ordered by hash. Functions with a unique hash value are easily eliminated. If the hash value matches the previous value or the next one, we must consider merging it. Otherwise it is dropped and never considered again.

The functions that remain are inserted into a binary tree, where functions are the node values themselves. An order relation is defined over the set of functions. We need total-ordering, so we need to maintain four properties on the functions set:

- $a <= a$ (reflexivity);
- if $a <= b$ and $b <= a$ then $a = b$ (antisymmetry);
- if $a <= b$ and $b <= c$ then $a <= c$ (transitivity);
- for all a and b, $a <= b$ or $b <= a$ (totality).

This total-ordering was made through special function comparison procedure that returns:

- 0 when functions are equal,
- -1 when Left function is less than right function, and
- 1 for opposite case.

This function comparison iterates through each instruction in each basic block. The FunctionComparator compares two functions to determine whether or not they will generate machine code with the same behaviour. DataLayout is used if available. The comparator always fails conservatively (erring on the side of claiming that two functions are different).

First it Compares the signature and other general attributes of the two functions. The functions must have identical signature, i.e., the same return type, the same number of arguments, and the same list of argument types, preserving argument order. Then they do a CFG-ordered walk since the actual ordering of the blocks in the linked list is immaterial. This walk starts at the entry block for both functions, then takes each block from each terminator in order. As an artifact, this also means that unreachable blocks are ignored. Finally, two blocks are equivalent if they have equivalent instructions in exactly the same order.

Functions are kept on binary tree. For each new function F we perform lookup in binary tree.

2.3 Optimisation Scope

2.3.1 Interprocedural Optimisations

2.3.2 Link-Time Optimisations

Chapter 3

Merging Pairs of Functions

In this section, we describe the proposed function-merging technique. Contrary to the state-of-the-art, our technique is able to merge any two functions. If the two functions are equivalent, i.e., identical, then the two functions can be completely merged into a single identical function. However, if the two functions differ at any point, an extra parameter is required, so that the caller is able to distinguish between the functions. The two functions can differ in any possible way, including their list of parameters or return types. If the lists of parameters are different, we can merge them so that we are able to uniquely represent all parameters from both functions. If the return types are different, we can use an aggregate type to return values of both types or return just the non-void type if the other one is void.

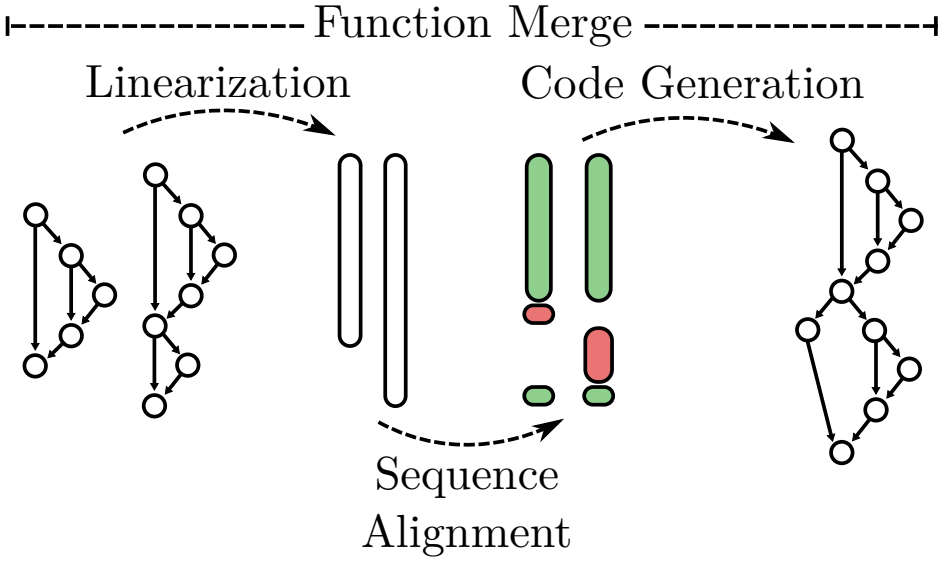


Figure 3.1: Overview of our function-merging technique.

The proposed technique consists of three major steps, as depicted in Figure 3.1.

First, we linearize each function, representing the CFG as a sequence of labels and instructions. The second step consists in applying a sequence alignment algorithm, borrowed from bioinformatics, which identifies regions of similarity between sequences. The sequence alignment algorithm allows us to arrange two linearized functions into segments that are equivalent between the two functions and segments where they differ from one another. The final step performs the code generation, actually merging the two functions into a single function based on the aligned sequences. Aligned segments with equivalent code are merged, avoiding redundancy, and the remaining segments where the two functions differ have their code guarded by a function identifier. At this point, we also create a merged list of parameters where parameters of the same type are shared between the functions, without necessarily keeping their original order. This new function can then be used to replace the original functions, as they are semantically equivalent, given the appropriate function-identifier parameter.

3.0.1 Linearization

The *linearization*[1] of a function consits in specifying an ordering of the basic blocks based on a traversal of the CFG and then producing a sequence of basic block labels and instructions, similar to a textual representation of the function. Although this operation is trivial, the specific ordering of the basic blocks chosen can have an impact on the merging operation.

In our implementation, the linearization uses a reverse post-order (RPO) of the basic blocks, following a canonical ordering of the successors. The RPO guarantees that the linearization starts with the entry basic block and then proceeds favoring definitions before uses, except in the presence of loops. Although the specifc ordering produced by the canonical linearization may not be optimal, it is common practice for compilers to rely on prior canonicalizations, e.g., canonical loops, canonical induction variables, canonical reassociation, etc. For contrast, if, instead, we use an RPO linearization with a uniformly randomized ordering of the successor basic blocks, the final code-size reduction of the function-merging optimization can drop up to 10% for individual benchmarks. Note that our decision for using the canonical RPO is purely pragmatic and other orderings of the basic blocks could also be used, as long as it produces a sequence of labels followed by instructions.

[1] Although linearization of CFGs usually refers to a predicated representation, in this paper, we refer to a simpler definition.

3.0.2 Sequence Alignment

When merging two functions, the goal is to identify which pairs of instructions and labels that can be merged and which ones need to be selected based on the actual function being executed. To avoid breaking the semantics of the original program, we also need to maintain the same order of execution of the instructions for each one of the functions.

To this end, after linearization, we reduce the problem of merging functions to the problem of *sequence alignment*. Sequence alignment is an important technique to many areas of science, most notably in molecular biology [**? ? ? ?**] where, for example, it is used for identifying homologous subsequences of amino acid in proteins. Figure 3.2 shows an example of the sequence alignment between two linearized functions extracted from the `400.perlbench` benchmark in SPEC CPU2006 [5].

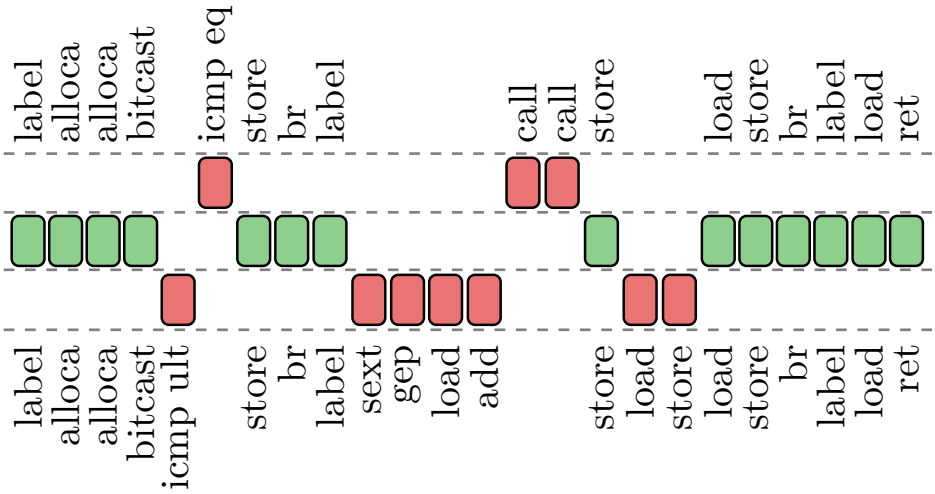


Figure 3.2: The sequence alignment between two functions.

Formally, sequence alignment can be defined as follows: For a given alphabet α, a sequence S of k characters is a subset of α^k, i.e., $S = (a_1, \ldots a_k)$. Let $S_1, \ldots, S_m$ be a set of sequences, possibly of different lengths but all derived from the same alphabet α, where $S_i = (a_1^{(i)}, \ldots, a_{k_1}^{(i)})$, for all $i \in \{1, \ldots, m\}$. Consider an extended alphabet that includes the *blank* character "$-$", i.e., $\beta = \alpha \cup \{-\}$. An alignment of the m sequences, $S_1, \ldots, S_m$, is another set of sequences, $\bar{S}_1, \ldots, \bar{S}_m$, such that each sequence $\bar{S}_i$ is obtained from S_i by inserting blanks in positions where some of the other sequences have non-blank and possibly equivalent characters, for a given equivalence relation. All sequences $\bar{S}_i$ in the alignment set have the same length l, where $\max\{k_1, \ldots, k_m\} \leq l \leq k_1 + \cdots + k_m$. Moreover, $\forall i \in \{1, \ldots, m\}$, $\bar{S}_i = (b_1^{(i)}, \ldots, b_l^{(i)})$,

there are increasing functions $v_i : \{1,\ldots,k_i\} \to \{1,\ldots,l\}$, such that:

- $b^{(i)}_{v_i(j)} = a^{(i)}_j$, for every $j \in \{1,\ldots,k_i\}$;
- any position j not mapped by the function v_i, i.e., for all $j \in \{1,\ldots,l\} \setminus \mathrm{Im}v_i$, then $b^{(i)}_j$ is a blank character.

Finally, for all $j \in \{1,\ldots,l\}$, there is at least one value of i for which $b^{(i)}_j$ is not a blank character. Note that two aligned sequences may contain both non-blank and non-equivalent characters at any given position, in which case it contains a mismatch.

Particularly for the function-merging, we are concerned with the alphabet consisting of all possible typed instructions and labels. Every linearized function represents a sequence derived from this alphabet. We explain the equivalence relation used for this alphabet in the next section.

There is a vast literature on algorithms for performing sequence alignment, especially in the context of molecular biology. These algorithms range from optimal algorithms based on dynamic programming to probabilistic models that does not guarantee optimality [**? ? ? ?**]. In this paper, we use the Needleman-Wunsh algorithm [**?**]. This algorithm is based on dynamic programming and consists of two main steps. First, it builds a *similarity matrix*, based on a scoring scheme, which assigns weights for matches, mismatches, and *gaps* (blank characters). Afterwards, a backward traversal is performed on the similarity matrix, in order to reconstruct the final alignment by maximizing the total score. We use a simple scoring scheme that rewards matches and penalizes mismatches and gaps.

3.0.3 Equivalence Relation

We describe the equivalence relation between values in two separate cases, namely, the equivalence between instructions and the equivalence between labels.

Labels can represent both normal basic blocks and landing blocks, which are used in exception handling code. Labels of normal basic blocks are always considered equivalent but landing blocks must have exactly the same landingpad instructions.

Two instructions are equivalent if: (1) their opcode are semantically equivalent, but not necessarily the same; (2) they both have equivalent types; and (3) they have pairwise operands with equivalent types. Types are considered equivalent if they can be bitcasted in a losslessly way from on to the other. It is also important to make sure

that there is no conflict regarding memory alignment when handling pointers. No additional restriction is imposed on the operands of the two instructions being compared for equivalence. Whenever two operands cannot be statically proved to represent the same value, a select instruction is used to distinguish between the execution of two functions being merged. For function calls, the type equivalence requires that both instructions have identical function types, i.e., both called functions must have an identical return type and an identical list of parameter types.

3.0.3.1 Handling Exception Handling Code

Most modern compilers implement the zero-cost Itanium ABI for exception handling [**?**], including GCC and LLVM, sometimes called the *landing-pad* model. In this section, we describe restrictions imposed by exception handling code and their equivalence relation.

The invoke instruction co-operates tightly with its landing block, i.e., the basic block pointed by the exception branch of an invoke instruction. The landing block must landingpad instruction as its first non-ϕ instruction. Given this restriction, two equivalent invoke instructions must also have landing blocks with equivalent landingpad instructions. This is easy to check since the landingpad instruction is always the first instruction in a landing block.

Landing blocks are responsible for handling all catch clauses of the higher-level programming language covering the particular callsite. All clauses are defined by the landingpad instruction, which encodes the list of all exception and cleanup handlers. Landingpad instructions are equivalent if they have the exactly same type and also encode an identical lists of exception and cleanup handlers. The type of equivalent landingpad instructions must be identical as its value is crucial in deciding what action to take when the landing block is entered, and corresponds to the return value of the personality function, which must also be identical for the two functions being merged.

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.

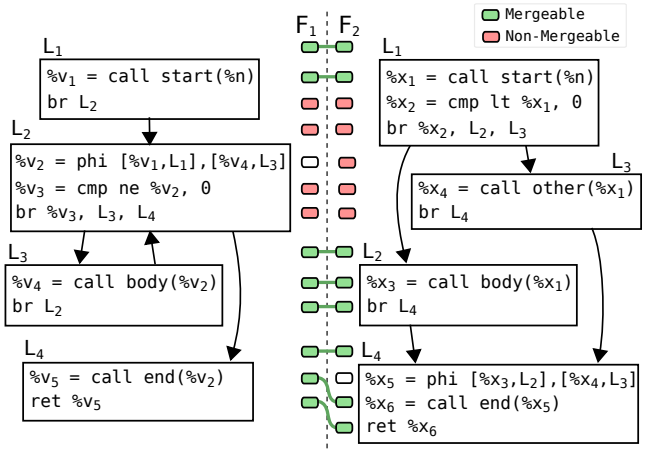


Figure 3.3: Example of functions aligned without register demotion. *Phi-nodes* are excluded from alignment.

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 3.0.4). Afterwards, all edges and operands are resolved, including appropriately assigning the incoming values to all *phi-nodes* (Section 3.0.5). SalSSA is designed to preserve all properties of SSA form via the standard SSA construction algorithm (Sections 3.0.6). Finally, SalSSA integrates a novel optimization with the SSA construction algorithm, called *phi-node coalescing*, producing even smaller merged functions (Section 3.0.7).

Working examples Figure 3.3 shows how the functions from our motivating example align without register demotion. Here, *phi-nodes* are not aligned, similarly to how FMSA 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.

3.0.4 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

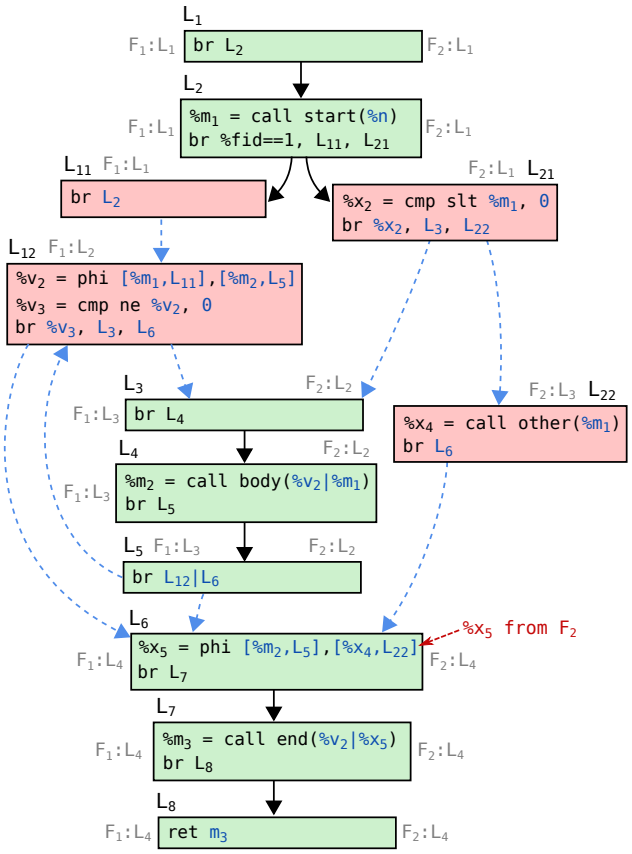


Figure 3.4: Merged CFG produced by SalSSA. Code corresponding to a single input basic block may be transformed into a chain of blocks, separating matching and non-matching code. The generator inserts conditional and unconditional branches to maintain the same order of instructions from the input basic block. Operands and edges highlighted in blue will be resolved by the operand assignment described in Section 3.0.5.

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 3.4 shows the generated CFG. At this point, the only instructions that actually have their operands assigned are the branches inserted to chain instructions orig-

inating 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 3.4, will be resolved later, during operand assignment.

3.0.4.1 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 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 3.4 shows an example where *phi-nodes* are present in basic blocks with both matching or non-matching labels. The phi-node $\texttt{x}_5$ is simply copied into the merged basic block labeled $\texttt{L}_6$.

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.

3.0.4.2 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 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 3.4. 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 3.0.5.3).

3.0.5 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 re-

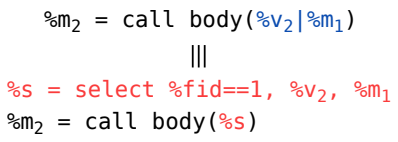


Figure 3.5: Operand selection for the `call` instruction in L_4 from Figure 3.4. Mismatching operands chosen with a `select` instruction on the function identifier.

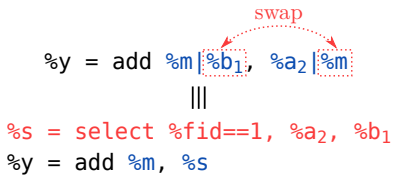


Figure 3.6: Optimizing operand assignment for commutative instructions. Example of a merged `add` instruction that can have its operands reordered to allow merging the two uses of `%m`, avoiding a *select* instruction.

maining edges in the control flow graph (dashed blue edges in Figure 3.4). 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 3.0.5.1 describes how we perform label selection. In all other cases, we simply use a *select* instruction, as shown in Figure 3.5.

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 3.6 shows an example of a commutative instruction where an operand selection can be avoided by reordering operands.

3.0.5.1 Label Selection

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

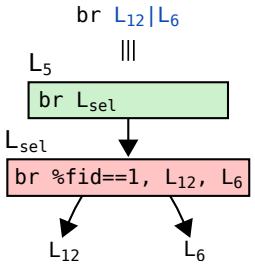


Figure 3.7: Label selection for mismatched terminator instruction operands L_{f1} and L_{f2} corresponding to labels of two different basic blocks. We handle control flow in a new basic block, L_{sel} with a conditional branch on the function identifier targeting the two labels. We use the label of the new block as the merged terminator operand.

labels. Then we use the new block's label as the operand of the merged terminator instruction. Figure 3.7 illustrates a CFG that handles label selection for a merged terminator instruction.

3.0.5.2 Landing Blocks

Most modern compilers, including GCC and LLVM, implement the zero-cost Itanium ABI for exception handling [**?**], 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 3.8.

3.0.5.3 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

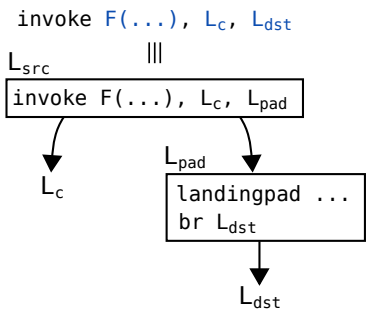


Figure 3.8: Landing blocks are added after operand assignment and are assigned to invoke instructions as operands.

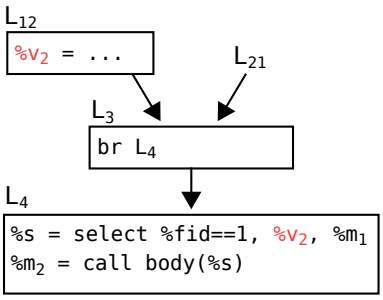
incoming flows present in the corresponding *phi-node* of their input function. For matching labels, each *phi-node* in the merged function will have additional incoming flows specific to the *other* input function but these flows should have undefined values.

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.

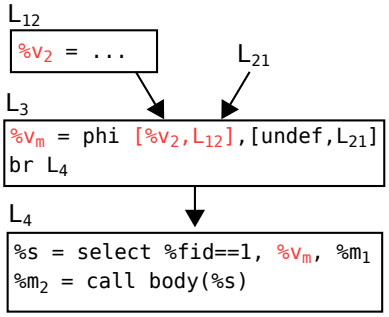
3.0.6 Preserving the Dominance Property

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 3.9a gives one example extracted from Figure 3.4 where the the dominance property is violated during code transformation.

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 [**?** **?**], 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



(a) Example where the dominance property is violated.



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

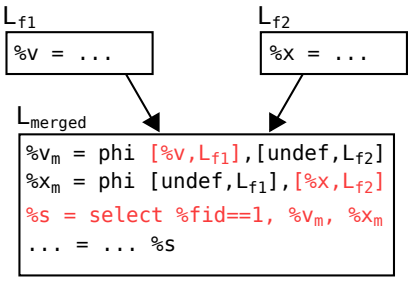
Figure 3.9: Example of how SalSSA uses the standard SSA construction algorithm to guarantee the dominance property of the SSA form.

names have a single definition by placing extra phi-nodes where needed so that instructions can be renamed appropriately. Figure 3.9b shows how the property violation in Figure 3.9a can be corrected using this strategy.

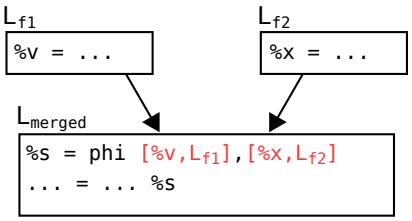
3.0.7 Phi-Node Coalescing

The approach described in Section 3.0.6 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 3.10 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 3.10a. This code inserts two trivial phi-nodes to select, again, `v` or `x` based on the executed



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 3.10: Phi-node coalescing reduces the number of phi-nodes and selections.

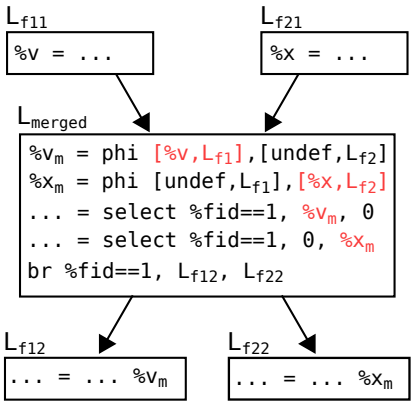
function. SalSSA optimizes this code by coalescing both phi-nodes into a single one and removing the selection statement. As shown in Figure 3.10b, the optimized version has a smaller number of instructions and phi-nodes.

This transformation is valid because a value definition that is exclusive to a function will never be used when executing the other function. Figure 3.11 shows another example illustrating that even disjoint definitions that have no user instructions in common can be coalesced, reducing the number of phi-nodes.

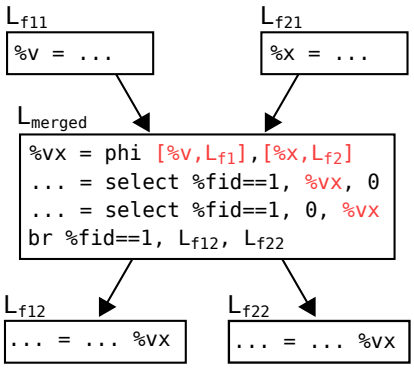
Since SalSSA is aware of which basic blocks are exclusive to each function, it can choose a pair of disjoint definitions for coalescing. Given a pair of disjoint definitions, SalSSA assigns the same name for both of them before applying the SSA reconstruction. SalSSA coalesces the set of definitions 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 re-



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 3.11: Reducing the number of phi-nodes by coalescing disjoint definitions with no user instructions in common.

duce code size. Consequently, it also enables more functions to be profitably merged.

Chapter 4

Function Merging Optimisation

4.1 Profitability Cost Model

After generating the code of the merged function, we need to estimate the code-size benefit of replacing the original pair of functions by the new merged function. In order to estimate the code-size benefit, we first compute the code-size cost for each instruction in all three functions. In addition to measuring the difference in size of the merged function, we also need to take into account all extra costs involved: (1) for the cases where we need to keep the original functions with a call to the merged function; and (2) for the cases where we update the call graph, there might be an extra cost with a call to the merged function due to the increased number of arguments.

Let $c(f)$ be the code-size cost of a given function f, and $\delta(f_i, f_j)$ represent the extra costs involved when replacing or updating function f_i with the function f_j. Therefore, given a pair of functions $\{f_1, f_2\}$ and the merged function $f_{1,2}$, we want to maximize the profit defined as:

$$\Delta(\{f_1, f_2\}, f_{1,2}) = (c(f_1) + c(f_2)) - (c(f_{1,2}) + \varepsilon)$$

where $\varepsilon = \delta(f_1, f_{1,2}) + \delta(f_2, f_{1,2})$. We consider that the merge operation is profitable if $\Delta(\{f_1, f_2\}, f_{1,2}) > 0$.

However, because we are operating on the IR level, one IR instruction does not necessarily translate to one machine instruction. Because of that, the profitability is measured with the help of the compiler's target-specific cost model. The actual cost of each instruction comes from querying this compiler's built-in cost model, which provides a target-dependent cost estimation that approximates the code-size cost of an IR instruction when lowered to machine instructions. Our implementation makes

use of the code-size costs provided by LLVM's target-transformation interface (TTI), which is widely used in the decision making of most optimizations [**? ?**].

4.2 Exhaustive Search

4.3 Focusing on Profitable Functions

Although the proposed technique is able to merge any two functions, it is not always profitable to merge them. In fact, as it is only profitable to merge functions that are sufficiently similar, for most pairs of functions, merging them increases code size. In this section, we introduce our framework for efficiently exploring the optimization space, focusing on pairs of functions that are profitable to merge.

For every function, ideally, we would like to try to merge it with all other functions and choose the pair that maximizes the reduction in code size. However, this quadratic exploration over all pairs of functions results in prohibitively expensive compilation overhead. In order to avoid the quadratic exploration of all possible merges, we propose the exploration framework shown in Figure 4.1 for our optimization.

The proposed framework is based on a light-weight ranking infrastructure that uses a *fingerprint* of the functions to evaluate their similarity. It starts by precomputing and caching fingerprints for all functions. The purpose of fingerprints is to make it easy to discard unpromising pairs of functions so that we perform the more expensive evaluation only on the most promising pairs. To this end, the fingerprint consists of: (1) a map of instruction opcodes to their frequency in the function; (2) the set of types manipulated by the function. While functions can have several thousands of instructions, an IR usually has just a few tens of opcodes, e.g., the LLVM IR has only about 64 different opcodes. This means that the fingerprint needs to store just a small integer array of the opcode frequencies and a set of types, which allows for an efficient similarity comparison.

By comparing the opcode frequencies of two functions, we are able to estimate the best case merge, which would happen if all instructions with the same opcode could match. This is a very optimistic estimation. It would be possible only if instruction types and order did not matter. We refine it further by estimating another best case merge, this time based on type frequencies, which would happen if all instructions with the same data type could match.

Therefore, the upper-bound reduction, computed as a ratio, can be generally de-

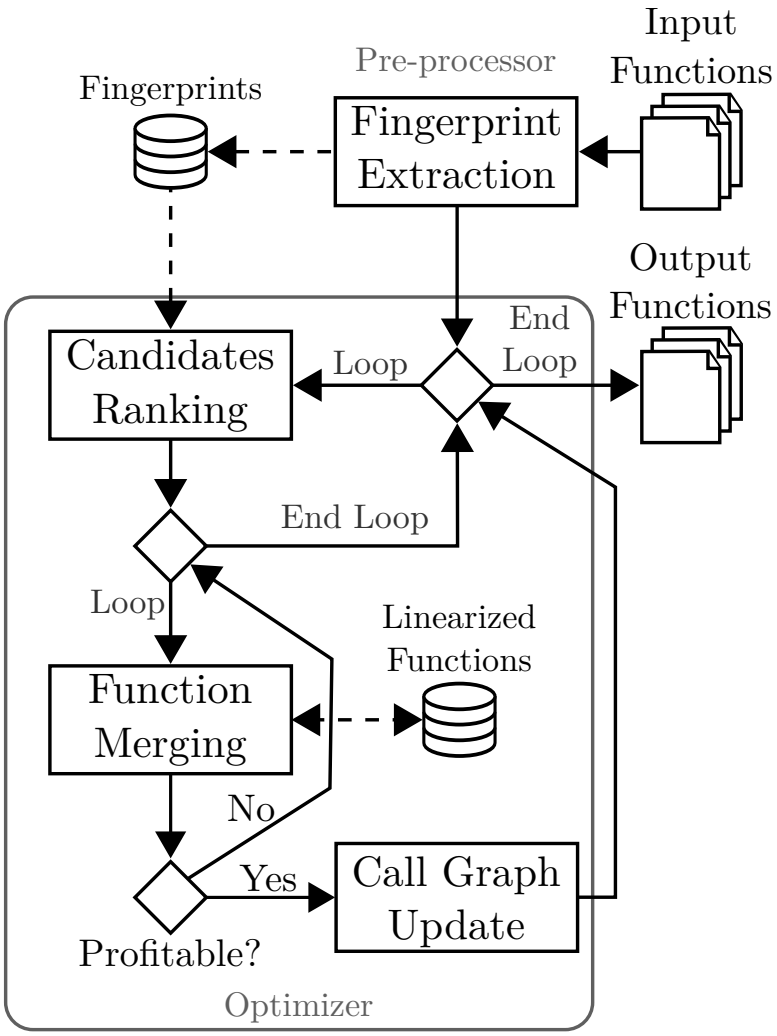


Figure 4.1: Overview of our exploration framework.

fined as

$$UB(f_1, f_2, K) = \frac{\sum_{k \in K} \min\{freq(k, f_1), freq(k, f_2)\}}{\sum_{k \in K} freq(k, f_1) + freq(k, f_2)}$$

where $UB(f_1, f_2, Opcodes)$ computes the opcode-based upper bound and $UB(f_1, f_2, Types)$ computes the type-based upper bound. The final estimate selects the minimum upper bound between the two, i.e.,

$$s(f_1, f_2) = \min\{UB(f_1, f_2, Opcodes), UB(f_1, f_2, Types)\}$$

This estimate results in a value in the range $[0, 0.5]$, which encodes a description that favors maximizing both the opcode and type similarities, while also minimizing their respective differences. Identical functions will always result in the maximum value of 0.5.

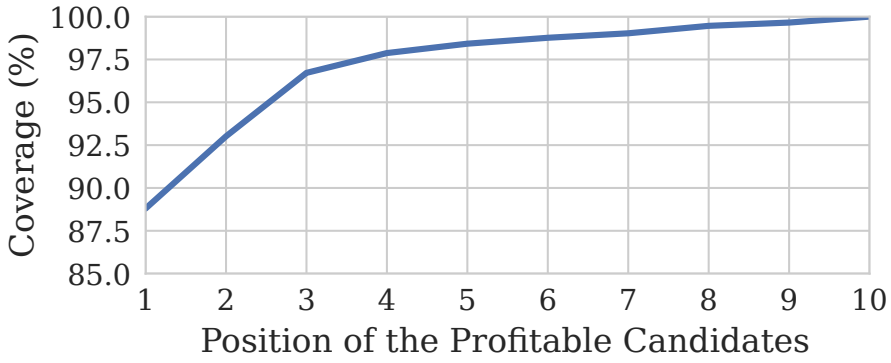


Figure 4.2: Average CDF for the position of the profitable candidate and the percentage of merged operations covered. 89% of the merge operations happen with the topmost candidate.

For each function f_1, we use a priority queue to rank the topmost similar candidates based on their similarity, defined by $s(f_1, f_2)$, for all other functions f_2. We use an exploration threshold to limit how many top candidates we will evaluate for any given function. We then perform this candidate exploration in a greedy fashion, terminating after finding the first candidate that results in a profitable merge and committing that merge operation.

Ideally, profitable candidates will be as close to the top of the rank as possible. Figure 4.2 shows the cumulative distribution of the position of the profitable candidates in a top 10 rank. It shows that about 89% of the merge operations occurred with the topmost candidate, while the top 5 cover over 98% of the profitable candidates. These results suggest that fingerprint similarity is able to accurately capture the real function similarity, while reducing the exploration cost by orders of magnitudes, depending on the actual number and size of the functions.

When a profitable candidate is found, we first replace the body of the two original functions to a single call to the merged function. Afterwards, if the original functions can be completely removed, we update the call graph, replacing the calls to the original functions by calls to the merged function. Finally, the new function is added to the optimization working list. Because of this feedback loop, merge operations can also be performed on functions that resulted from previous merge operations.

4.3.1 Link-Time Optimization

There are different ways of applying this optimization, with different trade-offs. We can apply our optimization on a per compilation-unit basis, which usually results in

lower compilation-time overheads because only a small part of the whole program is being considered at each moment. However, this also limits the optimization opportunities, since only pairs of functions within the same translation unit would be merged.

On the other hand, our optimization can also be applied in the whole program, for example, during link-time optimization (LTO). Optimizing the whole program is beneficial for the simple fact that the optimization will have more functions at its disposal. It allows us to merge functions across modules.

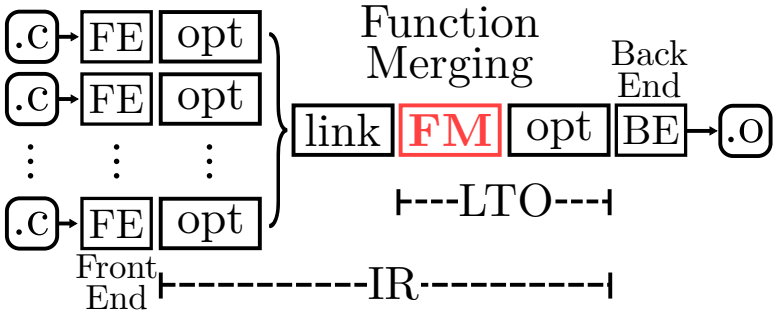


Figure 4.3: In our experiments we use a compilation pipeline with a monolithic link-time optimization (LTO).

In addition to the benefit of being able to merge more functions, when optimizing the whole program, we can also be more aggressive when removing the original functions, since we know that there will be no external reference to them. However, if the optimization is applied per translation unit, then extra conditions must be guaranteed, e.g., the function must be explicitly defined as internal or private to the translation unit.

Bibliography

[1] Microsoft Visual Studio. Identical COMDAT folding. https://msdn.microsoft.com/en-us/library/bxwfs976.aspx, 2020.

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

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

[4] Martin Liška. Optimizing large applications. *arXiv preprint arXiv:1403.6997*, 2014.

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

[6] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. Safe ICF: Pointer safe and unwinding aware identical code folding in gold. In *GCC Developers Summit*, 2010.