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

In recent years, the market for mobile and embedded systems has been rapidly growing. These systems often must run on inexpensive and resource-constrained devices, with limited memory, storage, CPU caches. and their applications are designed with different goals compared to traditional computing systems. Applications for these systems are designed with different goals compared to traditional computing systems.

Mobile systems need to support as many devices as possible, including low-end devices with limited resources [17, 22]. Another concern for mobile systems is the data consumption for downloads or updates of the application, which needs to be minimized as data over wireless carriers can be either limited or expensive. As a result, there is an increasing focus on the development of programs tailored for these low-end devices with limited memory sizes [1, 20].

In a similar way, embedded systems are becoming increasingly complex, with their application binaries often reaching several megabytes in size, turning memory size into a limiting factor [40]. Just adding more memory is not always a viable option. Highly integrated systems-on-chip are common in this market and their memories typically occupy the largest fraction of the chip area, contributing to most of the overall cost. Even small increases in memory area translate directly to equivalent increases in cost, which lead to enormous levels of lost profit at large scales [15].

In such constrained scenarios, reducing the application footprint is essential [6, 28, 43, 44, 49]. This is traditionally achieved by focusing on reducing the code size, either when designing the source code or by tuning the compiler to optimize for size [18, 26, 42, 44]. Despite the importance of keeping code size small, production compilers offer little help beyond a few classical optimisations [8, 10, 12]. Because of that, to avoid the expensive costs of extra storage and memory, the developers of these systems have to

manually find ways to shrink their code, which is also costly and undesirable [28, 51].

Code-size optimisations work by replacing a piece of code with another that is semantically equivalent but uses fewer or smaller instructions, sometimes combining and reusing equivalent pieces of code. Classical optimisations that are effective in reducing code size include the elimination of redundant, unreachable, and dead code, as well as certain kinds of strength reduction [8, 10, 12]. Although initially motivated by performance, these classical optimisations achieve better performance by focusing on code-size reduction.

One optimisation that can potentially reduce code size is function merging. In its simplest form, function merging reduces replicated code by combining multiple identical functions into a single one [3, 35]. This optimisation is found in linkers, by the name of *Identical code folding* (ICF), where text-identical functions at the bit level are merged [2, 32, 47]. However, such solutions are platform-specific and need to be adapted for each object code format and hardware architecture. Alternatively, compilers also provide a similar optimisation for merging identical functions at their mid-level intermediate representation (IR) and hence is agnostic to the target hardware [3, 35]. Unfortunately, these optimisations can only merge fully identical functions with at most type mismatches that can be losslessly cast to the same format.

More advanced approaches can identify similar 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. The work presented by von Koch et al. [16] proposed a merging strategy that exploits the isomorphism in the control-flow graphs (CFG) of the functions being merged. These functions can only differ between corresponding instructions, specifically, in their opcodes or the number and types of the input operands. However, they must have identical CFGs and function types.

Unfortunately, existing approaches fail to produce any noticeable code size reduction. In this work, we introduce a novel way to merge functions that overcomes major limitations of existing techniques. Our insight is that the weak results of existing function merging implementations are not due to the lack of duplicate code but due to the rigid, overly restrictive algorithms they use to find duplicates.

1.1 Contribution

Our approach is based upon the concept of sequence alignment, developed in bioinformatics for identifying functional or evolutionary relationships between different DNA or RNA sequences. Similarly, we use sequence alignment to find areas of functional similarity in arbitrary function pairs. Aligned segments with equivalent code are merged. The remaining segments where the two functions differ are added to the new function too but have their code guarded by a function identifier. This approach leads to significant code size reduction.

Applying sequence alignment to all pairs of functions is prohibitively expensive even for medium sized programs. To counter this, our technique is integrated with a ranking-based exploration mechanism that efficiently focuses the search to the most promising pairs of functions. As a result, we achieve our code size savings while introducing little compilation-time overhead.

Compared to identical function merging, we introduce extra code to be executed, namely the code that chooses between dissimilar sequences in merged functions. A naive implementation could easily hurt performance, e.g by merging two hot functions with only few similarities. Our implementation can avoid this by incorporating profiling information to identify blocks of hot code and effectively minimize the overhead in this portion of the code.

In this paper, we make the following contributions:

- We are the first to allow merging arbitrary functions, even ones with different signatures and CFGs.
- A novel ranking mechanism for focusing inter-procedural optimisations to the most profitable function pairs.
- Our function merging by sequence alignment technique is able to reduce code size by up to 25% on Intel and 30% on ARM, significantly outperforming the state-of-the-art, while introducing minimal compile-time and negligible run-time overheads.

Chapter 2

Background

2.1 Compiler Infrastructure

Compilers are programming tools responsible for translating programs in a given source language to a lower-level target language. This compilation process must preserve the program semantics. Moreover, compilers are also expected to produce a good quality representation of the program in the target language, optimising for a given objective function. An important objective function is code size, i.e., the optimisation goal is to produce a representation of the program as small as possible.

Compilers are usually organised in *three-phases*, as shown in Figure 2.1: frontend, optimiser, and backend. The frontend is responsible for parsing, validating and diagnosing errors in the source code. This parsed source code is then translated into an intermediate representation, which is the LLVM IR in this case. The optimiser is responsible for doing a broad variety of transformations, that are usually independent of language and target machine, to improve the code's performance. The backend, also known as the code generator, then translates the code from the intermediate representation onto the target instruction set. It is common for the backend to also perform some low-level optimisations that take advantage of unusual features of the supported architecture.

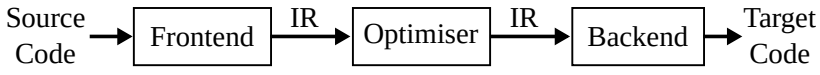


Figure 2.1: Overview of the three-phase compiler infrastructure.

However, these *three-phases* represent only a simplified view of their designed. In order to manage the complexity involved in optimising compilers, modern compilers

are usually designed in a highly modular manner, where they are organised as a series of phases that sequentially analyse and transform the program being compiled. For example, the frontend is subdivided into multiple phases. The *lexer* is responsible for tokenising the input stream of characters from the source code. This token stream is then consumed by the *parser*, producing a language-specific *Abstract Syntax Tree* (AST). The AST is an intermediate representation used during the semantic analysis, before being lowered to another intermediate representation.

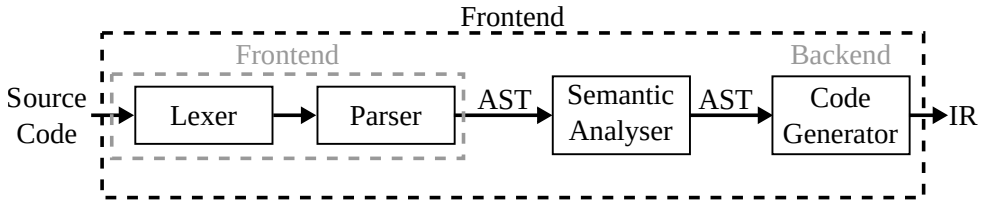
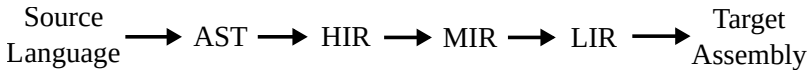
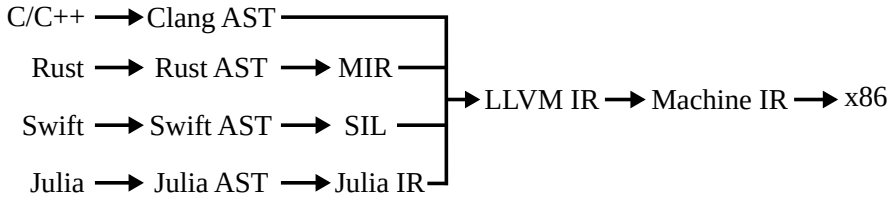


Figure 2.2: Breakdown of the frontend, illustrating how compilers are organised as a series of phases.



(a) An overview of representations and their level of abstractions used during the compilation pipeline.



(b) An example of the sequence of representations used in real compilers for different programming languages.

Figure 2.3: Sequence of representations used during the compilation pipeline in modern compilers.

Several intermediate representations, with different levels of abstraction, are used during this compilation process from the source to the target language. Different analyses and optimisations are better modelled at different abstraction levels. Figure 2.3a illustrates the sequence of representations used by modern compilers, each one having a progressively lower level than the previous one. The source language is parsed into an AST, which is then commonly lowered to a high-level intermediate representation (HIR). As shown in Figure fig:ir-lowering-sequence-example, this HIR is usually a

language-specific IR, such as the Swift Intermediate Language (SIL), which is used to solve domain-specific problems [33]. A popular mid-level intermediate representation (MIR) is the LLVM IR, which is shared among many compilers. In the LLVM compiler infrastructure, the LLVM IR is lowered into a low-level representation (LIR), called the Machine IR, which is then lowered into the target assembly language. This final process might actually involve multiple intermediate representations depending on the backend being used.

2.1.1 Link-Time Optimisations

Figure 2.4 illustrates the standard pipeline for the compilation of multiple source files. Compilers normally operate on a single translation unit at a time, where a translation unit includes a single source file and its expanded headers. Each translation unit is optimised separately and compiled into a single native object file. Finally, the linker combines multiple object files into a resulting binary or library.

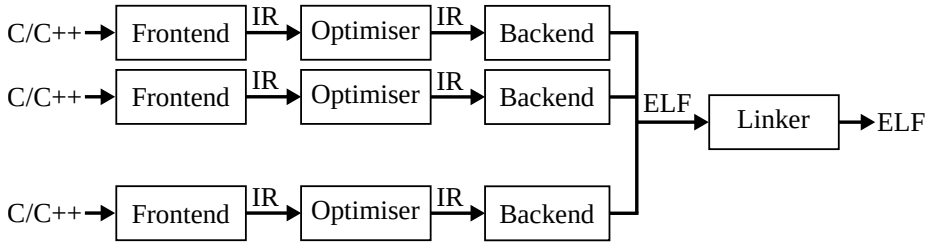


Figure 2.4: Overview of the three-phase compiler infrastructure.

However, this approach limits the impact of inter-procedural optimisations (IPO) to within each individual translation unit. For example, if we have two identical functions defined in different translation units, they will not be merged using the approach shown in Figure 2.4. In order to achieve larger benefits from IPO, the optimisation scope can be increased to include multiple translation units. When the scope includes all translation units being linked into an executable, the compiler can perform more aggressive optimisations that rely on whole-program information [27].

Figure 2.5 illustrates a common mechanism for enabling whole-program optimisation called link-time optimisation (LTO). In this approach, optimisations are applied in two different moments. First, we have *early* optimisations being applied on a per translation unit basis. However, we also have *late* optimisations being applied after all translation units are linked together. Therefore, allowing important inter-procedural optimisations to be applied on the whole-program, across translation units.

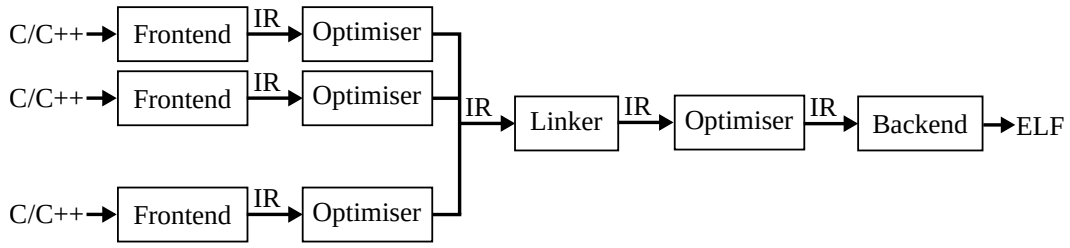


Figure 2.5: Overview of the three-phase compiler infrastructure.

In the LTO mode, compilers delay the generation of native object files. As shown in Figure 2.5, all translation units are linked while still in an IR better suited for the late optimisations.

2.2 Sequence Alignment

The comparison of two or more sequences, measuring the extent to which they differ, is important in many scientific areas, most notably in molecular biology [9, 39, 45, 50] where it has been critical in the understanding of functional, structural, or evolutionary relationships between the sequences [31, 37].

A particularly important comparison technique is sequence alignment, which identifies a series of patterns that appear in the same order in the sequences. Essentially, sequence alignment algorithms insert blank characters in both input sequences so that the final sequences end up having the same size, where equivalent segments are aligned with their matching segments from the other sequence and non-equivalent segments are either paired with the blank or a mismatching character.

Figure 2.6 shows an example of a pair-wise sequence alignment. This example, adapted from Lee et al. [34], shows two protein sequences where amino acids are represented by their one-letter symbology [4].

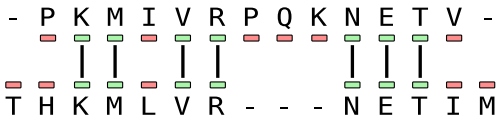


Figure 2.6: Example of an optimum alignment between two sequences. Matching segments are shown in green, vertically centred, and the non-matching segments are shown in red at the sides.

Formally, sequence alignment can be defined as follows: For a given alphabet α, a sequence S of k characters is an element 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 $S_i = (a_1^{(i)}, \ldots, a_{k_i}^{(i)})$, 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:

- $b_{v_i(j)}^{(i)} = a_j^{(i)}$, for every $j \in \{1, \ldots, k_i\}$;
- any position not covered by the function v_i contain a black character, i.e., for every $j \in \{1, \ldots, l\} \setminus \operatorname{Im} v_i$, b_j is the blank character "$-$".

Finally, for all $j \in \{1, \ldots, l\}$, there is at least one value of i for which $b_j^{(i)}$ 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 there is a mismatch.

The sequence alignment problem is concerned with identifying an alignment that maximises the score for a given scoring scheme. The scoring scheme first defines a weight for the alignment of pairs of characters which will then be used to compose a score for the whole sequence alignment. These weights are used to penalise mismatches and gaps while favouring matching pairs.

The alignment score between two characters is defined by a function on pairs of characters, $\delta \in \beta \times \beta \to \mathbb{R}$, for a given extended alphabet β. The simplest function that is commonly used is the constant function [21]. Let $a, b \in \beta$ and $a \neq b$. This constant function is defined by a triple $(w_1, w_2, w_3) \in \mathbb{R}^+ \times \mathbb{R}^- \times \mathbb{R}^-$, such that:

- For two matching caracters, $\delta(a, a) = w_1, w_1 \in \mathbb{R}^+$.
- For a mismatch betweem non-blank characters, $\delta(a, b) = w_2, w_2 \in \mathbb{R}^-$.
- The gap penalty, for when we have a blank character, $\delta(a, -) = \delta(-, a) = w_3, w_3 \in \mathbb{R}^-$.

This is a simple scoring scheme that rewards matches and penalises mismatches and gaps.

There is a vast literature on algorithms for performing sequence alignment, especially in the context of molecular biology. These algorithms are classified as either

global or local. A global sequence alignment algorithm attempts to align the entire sequence, using as many characters as possible, up to both ends of each sequence. Global alignment algorithms are useful for sequences that are highly similar and have approximately the same length [37]. Alternatively, a local sequence alignment algorithm generates subalignments in stretches of sequence with the highest density of matches. Local alignments are more suitable for aligning sequences with very few similarities or vastly different lengths [37].

In this work, we will focus on pair-wise global alignment algorithms. The following sections describe the main optimal algorithms based on dynamic programming. These algorithms will offer different optimality, performance, and memory usage trade-offs [9, 23, 39, 45].

2.2.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm [39] is one of the most well known algorithm for pair-wise global alignment. This algorithm gives an alignment that is guaranteed to be optimal for a given scoring scheme [24].

The Needleman-Wunsch 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.

		P	K	M	I	V	R	P	Q	K	N	E	T	V
	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13
T	-1	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-10	-11
H	-2	-2	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-11	-11
K	-3	-3	-1	-2	-3	-4	-5	-6	-7	-7	-8	-9	-10	-11
M	-4	-4	-2	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
L	-5	-5	-3	-1	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10
V	-6	-6	-4	-2	-2	0	-1	-2	-3	-4	-5	-6	-7	-8
R	-7	-7	-5	-3	-3	-1	1	0	-1	-2	-3	-4	-5	-6
N	-8	-8	-6	-4	-4	-2	0	0	-1	-2	-1	-2	-3	-4
E	-9	-9	-7	-5	-5	-3	-1	-1	-1	-2	-2	0	-1	-2
T	-10	-10	-8	-6	-6	-4	-2	-2	-2	-2	-3	-1	1	0
I	-11	-11	-9	-7	-5	-5	-3	-3	-3	-3	-3	-2	0	0
M	-12	-12	-10	-8	-6	-6	-4	-4	-4	-4	-4	-3	-1	-1

Figure 2.7: .

Figure 2.7 shows the similarity matrix corresponding to the example from Figure 2.6. The similarity matrix is constructed by comparing all possible pairs of characters from the input sequences. Let S_1 and S_2 be our input sequences of sizes k_1 and k_2, respectively, where $S_1 = (a_1, \ldots, a_{k_1})$ and $S_2 = (b_1, \ldots, b_{k_2})$. The similarity matrix M computed for these two input sequences will have size $(k_1+1) \times (k_2+1)$. Let $M_{i,j}$ denote all entries in the similarity matrix, with $1 \leq i \leq (k_1+1)$ and $1 \leq j \leq (k_2+1)$. The first entry in the matrix is $M_{1,1} = 0$, and

$$M_{i,j} = \max \begin{cases} M_{i-1,j} + \delta(a_{i-1}, -) & \text{if } i > 1 \text{ and } j \geq 1 \\ M_{i,j-1} + \delta(-, b_{j-1}) & \text{if } i \geq 1 \text{ and } j > 1 \\ M_{i-1,j-1} + \delta(a_{i-1}, b_{j-1}) & \text{if } i > 1 \text{ and } j > 1 \end{cases}$$

In other words, the score for each cell in the similarity matrix is the maximum among the rules shown in Figure 2.8.

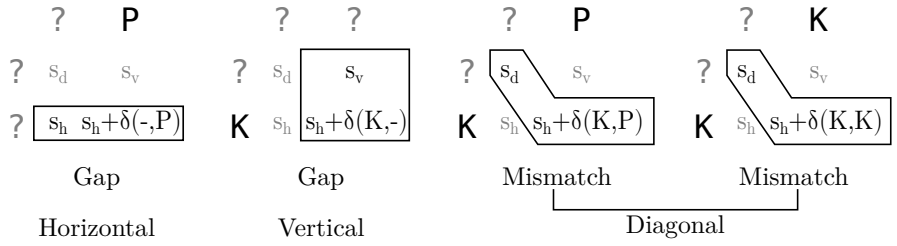


Figure 2.8: .

Figure 2.7 also highlights the traversal. Note that sometimes, while traversing the score matrix, there are multiple adjacent neighbours with the same score. Since there may exist multiple traversals with the same score, two sequences can have multiple optimum alignments.

Needleman-Wunsh algorithm is quadratic in the size of the sequences being aligned, both in time and space.

2.2.2 Hirschberg Algorithm

Chapter 3

Related Work

3.1 Code-Size Optimisations

Although initially motivated by performance, many of the classical optimisations achieve better performance by reducing code size. A small code, besides having fewer instructions to execute, can also have a positive impact on the cache utilisation. Classical optimisations that are effective in reducing code size include the elimination of redundant, unreachable, and dead code, as well as certain kinds of strength reduction [8, 10, 12]. In this section, we will describe some of these classical size-reducing optimisations.

3.1.1 Unreachable-Code Elimination

Some functions may contain code that is unreachable. A code is unreachable if there is no valid control-flow path from the function's entry point that leads to it. Since unreachable code is guaranteed to never be executed, compilers should remove it to avoid code bloat.

Often, unreachable code is uncovered by other optimizations. For example, after constant propagation, a conditional branch could have its condition evaluating to a constant, eliminating a path to one of its successor basic block. If no other path leads to that basic block, it becomes unreachable.

The algorithm to eliminate unreachable code works in a mark-sweep manner, performing two passes over the basic blocks of the CFG. The reachability analysis optimistically assumes that all basic blocks are dead until proven otherwise. First, it marks all blocks as unreachable. Next, starting from the entry point, it marks each block that it can reach as reachable. If all branches and jumps are unambiguous, then all

unmarked blocks can be deleted. With ambiguous branches or jumps, the compiler must preserve any block that the branch or jump can reach. This analysis is simple and inexpensive.

3.1.2 Dead-Code Elimination

3.2 Merging Identical Functions

In this section, we will discuss existing optimisations for merging identical functions. Figure 3.1 illustrates how identical functions can appear in real programs. The first pair of functions, shown in Figure 3.1a, were extracted from the `482.sphinx3` benchmark. The only difference between these two functions is in their parameter type. However, all pointer types can be considered equivalent since they can be bitcasted in a losslessly way. These functions are usually produced by copy-and-paste programming, where a given code pattern is copied and then repurposed [5, 25, 29]. The second pair of functions, shown in Figure 3.1b, were extracted from the `403.gcc` benchmark and they are fully identical. These functions are part of GCC's backend, where it is common to have code that is automatically generated from a machine description [19, 30, 38].

```
void delete_contexts_MotionInfo
     (MotionInfoContexts *enco_ctx)
{
  if( enco_ctx == NULL )
    return;

  free( enco_ctx );

  return;
}
```

```
void delete_contexts_TextureInfo
     (TextureInfoContexts *enco_ctx)
{
  if( enco_ctx == NULL )
    return;

  free( enco_ctx );

  return;
}
```

(a) Two semantically identical functions extracted from the `482.sphinx3` benchmark.

```
rtx
gen_floathitf2 (operand0, operand1)
     rtx operand0;
     rtx operand1;
{
  return gen_rtx_SET (VOIDmode,
         operand0,
         gen_rtx_FLOAT (TFmode,
         operand1));
}
```

```
rtx
gen_floatsitf2 (operand0, operand1)
     rtx operand0;
     rtx operand1;
{
  return gen_rtx_SET (VOIDmode,
         operand0,
         gen_rtx_FLOAT (TFmode,
         operand1));
}
```

(b) Two semantically identical functions extracted from the `403.gcc` benchmark.

Figure 3.1: Example of identical functions.

Note, however, that functions can be identical at the IR or machine level without necessarily being identical at the source level. Figure 3.2 shows two real functions

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

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

After template specialization and applying optimizations:

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

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

Figure 3.2: Two function extracted from the `447.dealII` benchmark that are not identical at the source level, but after applying template specialisation and optimisations they become identical at the IR level.

extract from the 447.dealII program in the SPEC CPU2006 [46] benchmark suite. Although these two functions are not identical at the source level, they become identical after a template specialisation and some optimisations 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 simplify the loops in both functions, resulting in the identical functions shown at the bottom of Figure 3.2.

Identical code is particularly common in C++ programs with heavy use of *parametric polymorphism*, via template or *auto* type deduction.

3.2.1 Merging Identical Object Code During Link Time

The simplest way of merging identical functions is by looking at their object code, during link time. *Identical code folding* (ICF) is an optimisation that identifies and merges two or more read-only sections, typically functions, that have identical contents. This optimisation is commonly found in major linkers, such as *gold* [32, 47], LLVM's *lld*, and the MSVC linker [2].

Before applying ICS, it is a common practice for linkers to place functions in separate sections [32, 47]. Therefore, merging identical functions can be generalised to the problem of merging identical sections. In the simplest case, two functions or sections

are identical if they have exactly the same binary representation. In general, two sections are considered identical if they have the identical section flags, data, code, and relocations. Two relocations are considered identical if they have the same relocation types, values, and if they point to the same or, recursively, identical sections.

Once a set of identical functions have been identified, merging them requires a simple operation. Only one of the functions in the set is kept for the final binary and all the other copies are discarded. Afterwards, every reference to the discarded functions must be redirected to the kept function.

Since the equality relation has a cyclic definition, ICF is defined as a fixed-point computation, i.e., it is applied repeatedly until a convergence is obtained. There are two approaches with distinct trade-offs: (i) The pessimistic approach starts with all sections marked as being different and then repeatedly compare them trying to prove their equality, grouping those found to be identical, including their relocations. This approach is implemented in the widely used *gold* linker. (ii) The optimistic approach starts with all functions marked as potentially identical and then repeatedly compare trying to disprove their equality, partitioning those found to be different. This approach is implemented in LLVM's linker, *lld*.

3.2.2 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 [3, 35]. This optimisation is only flexible enough to accommodate simple type mismatches provided they can be bitcasted in a losslessly way.

A very strict function comparator is used to identify if two functions are semantically equivalent. 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 exactly the same list of argument types. Then this function comparator performs a simultaneous walk, in depth first order, in the functions' control-flow graphs. 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, it iterates through each instruction in each basic block. Two blocks are equivalent if they have equivalent instructions in exactly the same order, without excess. The comparator always fails conservatively, erring on the side of claiming that two functions are different.

When a pair of equivalent functions is identified, we can create either an alias or a *thunk*. Aliasing entails eliminating one of the functions and replacing all its call-sites to the other function. Thunks must be created when neither of the equivalent functions can be eliminated by aliasing. In such case, a thunk is created for either one of the functions, replacing its body by a call to the other function, which allows all call-sites and name references to both functions to be preserved. Aliasing is preferred since it is cheaper and adds no runtime overhead. The appropriate merging is applied according to following rules:

- If the address of at least one function is not taken, alias can be used.
- But if the function is part of COMDAT section that can be replaced, we must use thunk.
- If we create a thunk and none of functions is writeable, we can redirect calls instead.

Although very restrictive, this optimisation guarantees that any pair of mergeable functions will result in code size reduction with no performance overhead.

Its simplicity also allows for an efficient exploration approach based on computing a hash of the functions and then using a binary tree to identify equivalent functions. Since hashing is cheap to compute, it allows us to efficiently group possibly equivalent functions and filter out functions that are obviously unique. This hash must have 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.

A function hash is calculated by considering only the number of arguments and whether a function contains variadic parameters, the order of basic blocks (given by the successors of each basic block in depth first order), and the order of opcodes of each instruction within each of these basic blocks. This mirrors the strategy of compare() uses to compare functions by walking the BBs in depth first order and comparing each instruction in sequence. Because this hash does not look at the operands, it is insensitive to things such as the target of calls and the constants used in the function, which makes it useful when possibly merging functions which are the same modulo constants and call targets.

All functions can be sorted based on their hash value, which ends up grouping possibly equivalent functions together. If the hash value of a given function matches any of its adjacent values in the sorted list, this function must be considered for merging. Functions with a unique hash value can be easily ignored since no other function will be found equivalent.

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

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

3.3 Merging Beyond Identical Functions

In the previous sections, we have seen compiler optimisations that merge identical functions. However, nearly identical functions, with only minor differences, are also commonly found. Figure 3.3 shows two examples of nearly identical functions found in real programs. The highlighted differences prevent these functions from being merged by the identical function merging techniques. The first pair of functions, shown in Figure 3.3a, illustrates code that is usually produced by copy-and-paste programming, where a given code pattern is copied and then repurposed [5, 25, 29]. The

```
double DMin(double *vec, int n) {
  int i;
  double best;
  best = vec[0];
  for (i=1; i<n; i++)
    if (vec[i] < best)
      best = vec[1];
  return best;
}
```

```
double DMax(double *vec, int n) {
  int i;
  double best;
  best = vec[0];
  for (i=1; i<n; i++)
    if (vec[i] > best)
      best = vec[1];
  return best;
}
```

(a) Two similar functions extracted from the `456.hmmer` benchmark.

```
rtx
gen_peephole2_1270 (curr_insn, operands)
     rtx curr_insn ATTRIBUTE_UNUSED;
     rtx *operands;
{
  rtx _val = 0;
  HARD_REG_SET _regs_allocated;
  CLEAR_HARD_REG_SET (_regs_allocated);
  start_sequence ();
  operands[2] = GEN_INT(
      exact_log2(INTVAL(operands[1])));
  emit (gen_rtx_PARALLEL (VOIDmode,
        gen_rtvec (2,
                gen_rtx_SET (VOIDmode,
        operands[0],
        gen_rtx_ASHIFT (SImode,
        copy_rtx (operands[0]),
        operands[2])),
                gen_rtx_CLOBBER (VOIDmode,
        gen_rtx_REG (CCmode,
        17)))));
  _val = gen_sequence ();
  end_sequence ();
  return _val;
}
```

```
rtx
gen_peephole2_1271 (curr_insn, operands)
     rtx curr_insn ATTRIBUTE_UNUSED;
     rtx *operands;
{
  rtx _val = 0;
  HARD_REG_SET _regs_allocated;
  CLEAR_HARD_REG_SET (_regs_allocated);
  start_sequence ();
  operands[2] = GEN_INT(
      exact_log2(INTVAL(operands[1])));
  emit (gen_rtx_PARALLEL (VOIDmode,
        gen_rtvec (2,
                gen_rtx_SET (VOIDmode,
        operands[0],
        gen_rtx_ASHIFT (DImode,
        copy_rtx (operands[0]),
        operands[2])),
                gen_rtx_CLOBBER (VOIDmode,
        gen_rtx_REG (CCmode,
        17)))));
  _val = gen_sequence ();
  end_sequence ();
  return _val;
}
```

(b) Two similar functions extracted from the `403.gcc` benchmark.

Figure 3.3: Example of

second pair of functions, shown in Figure 3.3b, are produced by generative programming [11, 13], where their code was automatically generated using a description language [19].

Von Edler et al. [16] have proposed a function-merging technique exploits structural similarity among functions. Their optimization is able to merge similar functions that are not necessarily identical. Two functions are structurally similar if both their function types are equivalent and their CFGs isomorphic. Two function types are equivalent if they agree in the number, order, and types of their parameters as well as their return types, linkage type, and other compiler-specific properties. In addition to the structural similarity of the functions, their technique also requires that corresponding basic blocks have exactly the same number of instructions and that corresponding instructions must have equivalent resulting types. Mergeable functions are only allowed to differ in corresponding instructions, where they can differ in their opcodes or in the number and type of their input operands.

Because the state-of-the-art is limited to functions with identical CFGs and function types, once it merges a pair of functions, a third *similar* function cannot be merged into the resulting merged function since they will differ in both CFGs and their lists of parameters. Due to this limiting factor, the state-of-the-art has to first collect all mergeable functions and merge them simultaneously.

The state-of-the-art algorithm iterates simultaneously over corresponding basic blocks in the set functions being merged, as they have isomorphic CFGs. For every basic block, if their corresponding instructions have different opcodes, they split the basic block and insert a switch branch to select which instruction to execute depending on a function identifier. Because these instructions have equivalent resulting types, their results can be merged using a phi-operator, which can then be used transparently as operands by other instructions.

Although the state-of-the-art technique improves over LLVM's identical function merging, it is still unnecessarily limited. In Section **??**, we showed examples of very similar real functions where the state-of-the-art fails to merge. Our approach addresses such limitations improving on the state-of-the-art across the board.

TODO: Comparison table: Identical vs VonKoch vs Ours

3.4 Code Factoring

Code factoring is a related technique that addresses the same fundamental problem of duplicated code in a different way. Code factoring can be applied at different levels of the program [36]. Local factoring, also known as local code motion, moves identical instructions from multiple basic blocks to either their common predecessor or successor, whenever valid [7, 36**?**]. Procedural abstraction finds identical code that can be extracted into a separate function, replacing all replicated occurrences with a function call [14, 36].

Procedural abstraction differs from function merging as it usually works on single basic blocks or single-entry single-exit regions. Moreover, it only works for identical segments of code, and every identical segment of code is extracted into a separate new function. Function merging, on the other hand, works on whole functions, which can be identical or just partially similar, producing a single merged function.

However, all these techniques are orthogonal to the proposed optimization and could complement each other at different stages of the compilation pipeline.

3.5 Code Similarity

Code similarity has also been used in other compiler optimizations or tools for software development and maintenance. In this section, we describe some of these applications.

Coutinho et al. [**?**] proposed an optimization that uses instruction alignment to reduce divergent code for GPU. They are able to fuse divergent branches that contain single basic blocks, improving GPU utilization.

Similarly, analogous algorithms have also been suggested to identify the differences between two programs, helping developers with source-code management and maintenance [**?** **?**]. These techniques are applied in tools for source-code management, such as the *diff* command [**?**].

Similar techniques have also been applied to code editors and IDEs [**?** **?**]. For example, SourcererCC [**?**] detects possible clones, at the source level, by dividing the programs into a set of code blocks where each code block is itself represented by a bag-of-tokens, i.e., a set of tokens and their frequencies. Tokens are keywords, literals, and identifiers, but not operators. Code blocks are considered clones if their degree of similarity is higher than a given threshold. In order to reduce the number of blocks compared, candidate blocks are filtered based on a few of their tokens where at least one must match.

Our ranking mechanism uses an approach similar to SourcererCC, where we use opcode frequencies and type frequencies to determine if two functions are likely to have similar code. However, we need a precise and effective analysis of code similarity when performing the actual merge. To this end, we use a sequence alignment technique.

Chapter 4

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.

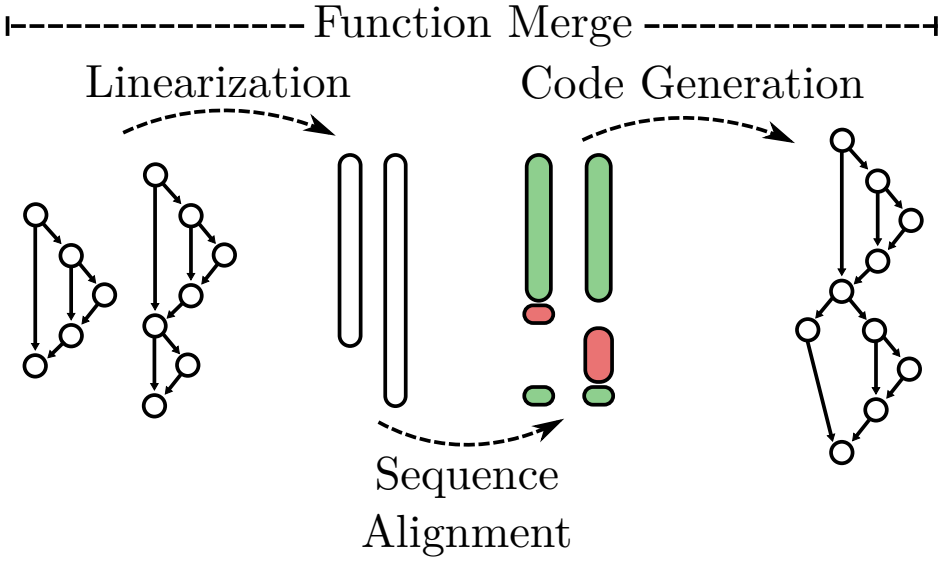


Figure 4.1: Overview of our function-merging technique.

The proposed technique consists of three major steps, as depicted in Figure 4.1.

First, we linearise 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 linearised 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.

4.0.1 Linearisation

The *linearisation*[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 linearisation uses a reverse post-order (RPO) of the basic blocks, following a canonical ordering of the successors. The RPO guarantees that the linearisation 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 linearisation may not be optimal, it is common practice for compilers to rely on prior canonicalisations, e.g., canonical loops, canonical induction variables, canonical reassociation, etc. For contrast, if, instead, we use an RPO linearisation with a uniformly randomised ordering of the successor basic blocks, the final code-size reduction of the function-merging optimisation 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 linearisation of CFGs usually refers to a predicated representation, in this paper, we refer to a simpler definition.

4.0.2 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 linearisation, we reduce the problem of merging functions to the problem of *sequence alignment*. Figure 4.2 shows an example of the sequence alignment between two linearised functions extracted from the `400.perlbench` benchmark in SPEC CPU2006 [46].

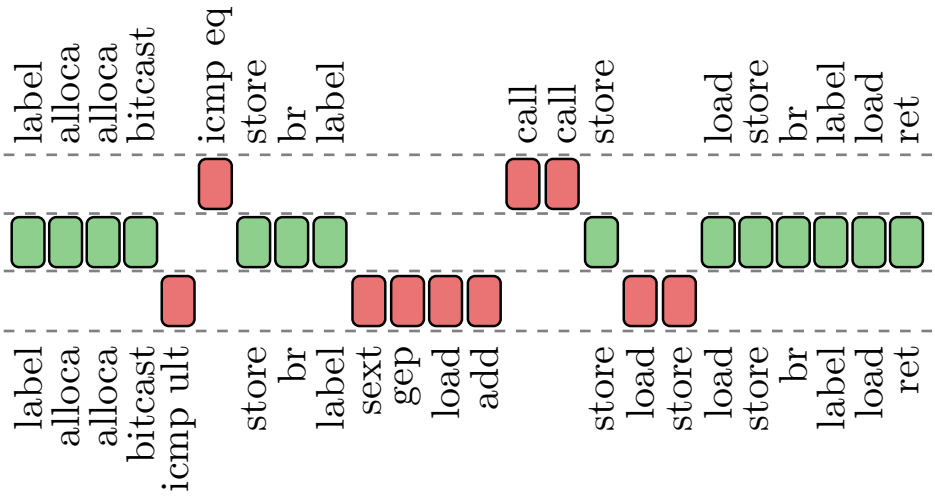


Figure 4.2: The sequence alignment between two functions.

Specifically for the function merging, we are concerned with the alphabet consisting of all possible typed instructions and labels. Every linearised function represents a sequence derived from this alphabet. We explain the equivalence relation used for this alphabet in the next section. Although we only consider pair-wise alignments, the technique would also work for multi-sequences.

4.0.3 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.

4.0.3.1 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.

4.1 Code Generation

The code generation phase is responsible for producing a new function from the output of the sequence alignment. Our four main objectives are: merging the parameter lists; merging the return types; generating select instructions to choose the appropriate operands in merged instructions; and constructing the CFG of the merged function.

Our approach can effectively handle multiple different function merging scenarios:

- identical functions,
- functions with differing bodies,
- functions with different parameter lists,
- functions with different return types,
- and any combination of these cases.

4.1.0.1 Merged Parameters

To maintain the semantics of the original functions, we must be able to pass their parameters to the new merged function. The merged parameter list is the union of the original lists, with placeholders of the correct type for any of the parameters. Maintaining the original order is not important for maintaining semantics, so we make no effort to do so. If the two functions have differing bodies, we add an extra binary parameter, called the function identifier, to the merged list of parameters. This extra parameter is required for selecting code that should be executed only for one of the merged functions.

Figure 4.3 depicts how we merge the list of parameters of two functions. First, we create the binary parameter that represents the function identifier, one of the functions will be identified by the value `true` and the other by the value `false`. We then add all the parameters of one of the functions to the new list of parameters. Finally, for each parameter of the second function, we either reuse an existing and available parameter of identical type from the first function or we add a new parameter. We keep track of the mapping between the lists of parameters of the original functions and the merged function so that, later, we are able to update the function calls. When replacing the function calls to the new merged function, parameters that are not used by the original function being called will receive undefined values.

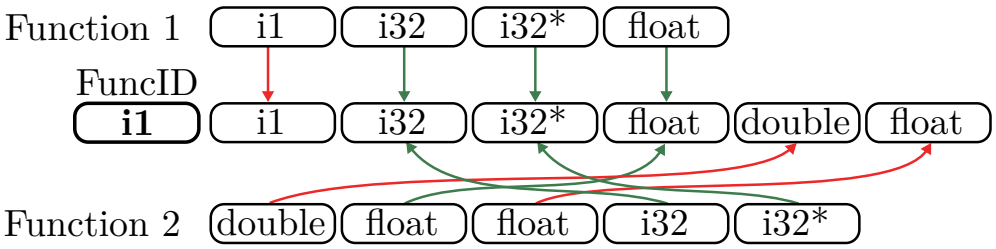


Figure 4.3: Example of a merge operation on the parameter lists of two functions.

The reuse of parameters between the two merged functions provides the following benefits: (1) it reduces the overheads associated with function call abstractions, such as reducing the number of values required to be communicated between functions. (2) if both functions use merged parameters in similar ways, it will remove some of the cases where we need select instructions to distinguish between the functions.

There are multiple valid ways of merging parameter lists. For example, multiple parameters of one function may have the same type as a given parameter from the other function. In such cases, we select parameter pairs that minimize the number of select instructions. We find them by analyzing all pairs of equivalent instruction that use the parameters as operands. Our experiments show that maximizing the matching of parameters, compared to never merging them, improves code-size reduction of individual benchmarks by up to 7%.

4.1.0.2 Control-Flow Graph Reconstruction

Our technique is able to merge any return types. When merging return types, we select the largest one as the base type. Then, we use bitcast instructions to convert between the types. Before a return instruction, we bitcast the values to the base return type. We reverse this at the call-site, where we cast back to the original type. Having identical types or void return are just special cases where casting is unnecessary. In the case of void types, we can just return *undefined* values since they will be discarded at the corresponding call-sites.

After generating the merged list of parameters and return type, we produce the CFG of the merged function.

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

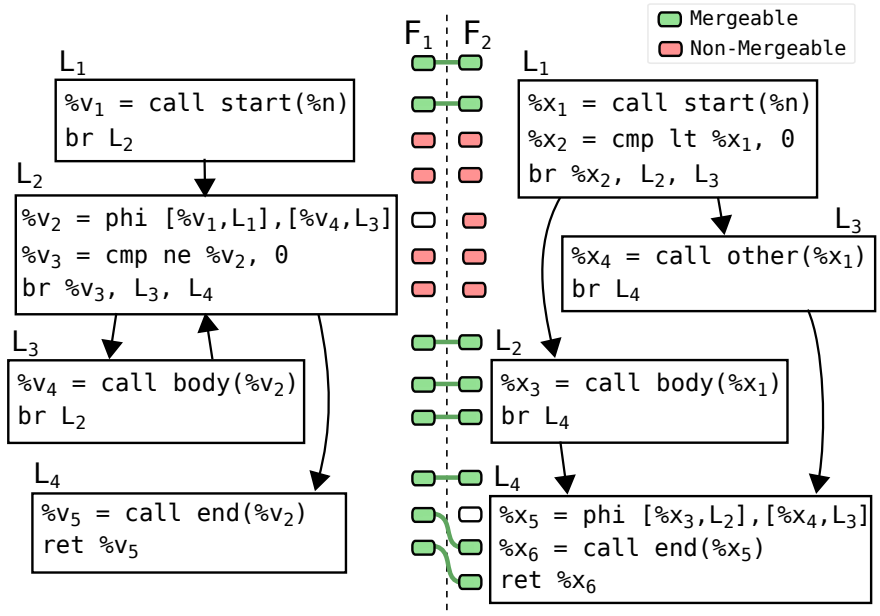


Figure 4.4: Example of functions aligned without register demotion. *Phi-nodes* are excluded from alignment.

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 4.1.1). Afterwards, all edges and operands are resolved, including appropriately assigning the incoming values to all *phi-nodes* (Section 4.1.2). SalSSA is designed to preserve all properties of SSA form via the standard SSA construction algorithm (Sections 4.1.3). Finally, SalSSA integrates a novel optimisation with the SSA construction algorithm, called *phi-node coalescing*, producing even smaller merged functions (Section 4.1.4).

Working examples Figure 4.4 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.

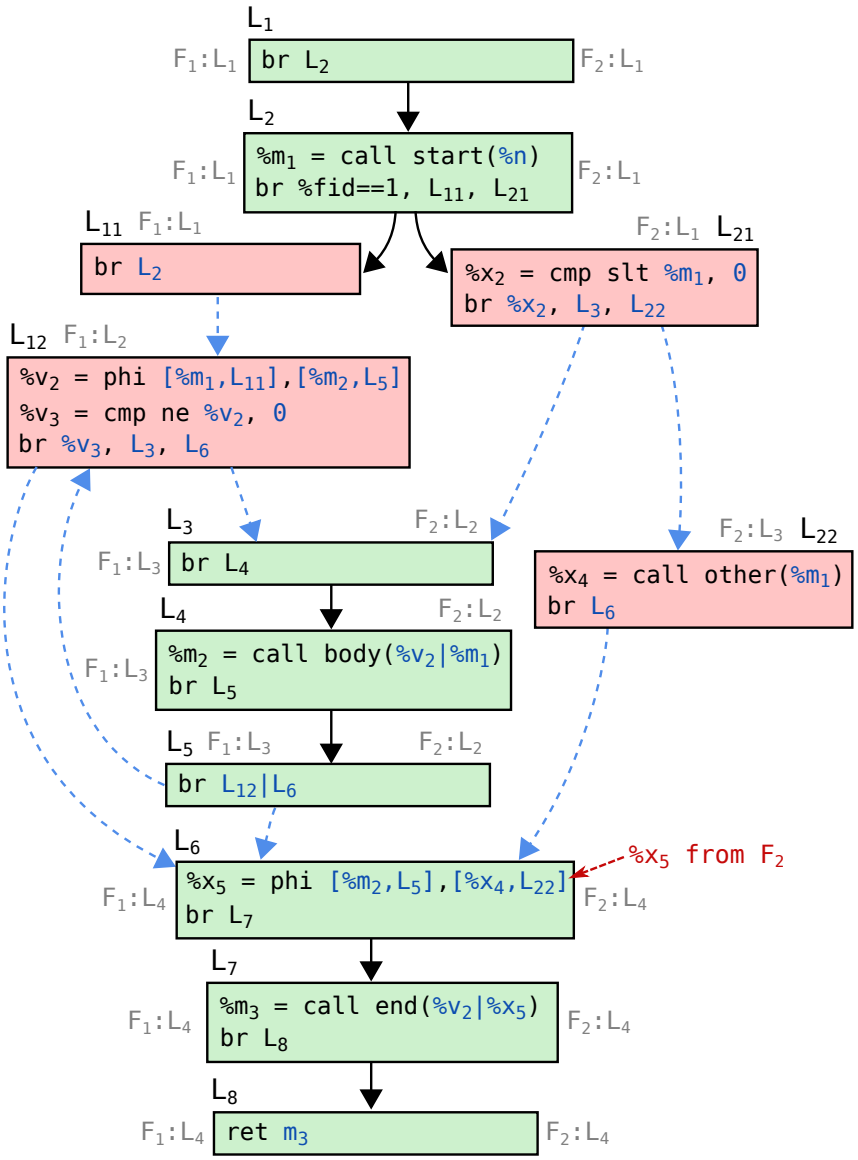


Figure 4.5: Merged CFG produced by SalSSA. Code corresponding to a single input basic block may be transformed into a chain of blocks, separating matching and non-matching code. The generator inserts conditional and unconditional branches to maintain the same order of instructions from the input basic block. Operands and edges highlighted in blue will be resolved by the operand assignment described in Section 4.1.2.

4.1.1 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 4.5 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 4.5, will be resolved later, during operand assignment.

4.1.1.1 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 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 4.5 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 optimisations from LLVM.

4.1.1.2 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

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

Figure 4.6: Operand selection for the `call` instruction in L_4 from Figure 4.5. Mismatching operands chosen with a `select` instruction on the function identifier.

```
                 swap
%y = add %m|%b1, %a2|%m
           |||
%s = select %fid==1, %a2, %b1
%y = add %m, %s
```

Figure 4.7: 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.

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 4.5. 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 4.1.2.3).

4.1.2 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 4.5). 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 4.1.2.1 describes how we perform label selection. In all other cases, we simply use a *select* instruction, as shown in Figure 4.6.

When assigning operands to commutative instructions, we also perform operand reordering to maximise the number of matching operands and reduce the need for

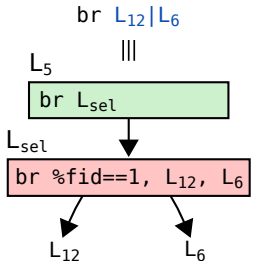


Figure 4.8: Label selection for mismatched terminator instruction operands L_{f1} and L_{f2} corresponding to labels of two different basic blocks. We handle control flow in a new basic block, L_{sel} with a conditional branch on the function identifier targeting the two labels. We use the label of the new block as the merged terminator operand.

select instructions. Figure 4.7 shows an example of a commutative instruction where an operand selection can be avoided by reordering operands.

4.1.2.1 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 labels. Then we use the new block's label as the operand of the merged terminator instruction. Figure 4.8 illustrates a CFG that handles label selection for a merged terminator instruction.

4.1.2.2 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

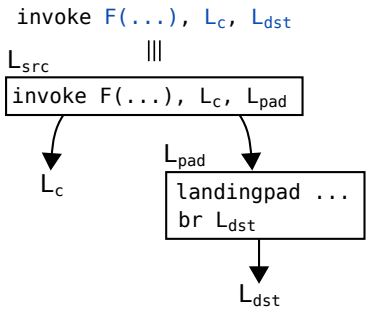


Figure 4.9: Landing blocks are added after operand assignment and are assigned to invoke instructions as operands.

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 4.9.

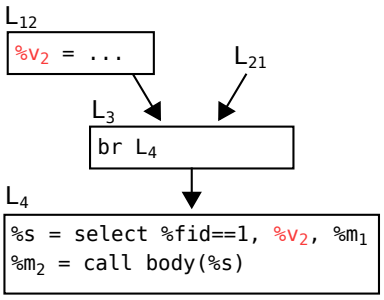
4.1.2.3 Phi-Node's Incoming Values

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.

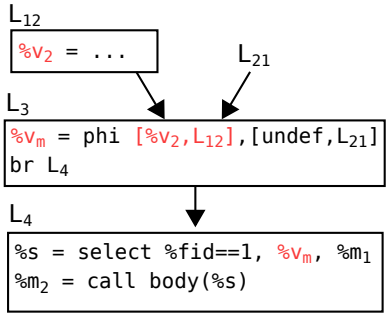
To assign the incoming values of a *phi-node*, SalSSA iterates over all predecessors of its parent basic block and uses the *block mapping* to discover each predecessor's corresponding basic block in the input function. If such a basic block is found, then SalSSA obtains the incoming value associated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

4.1.3 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



(a) Example where the dominance property is violated.

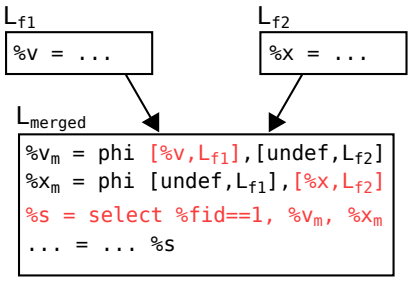


(b) The dominance property is restored by placing phi-nodes where needed.

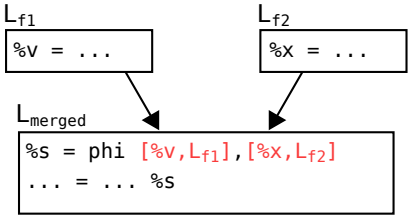
Figure 4.10: Example of how SalSSA uses the standard SSA construction algorithm to guarantee the dominance property of the SSA form.

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 4.10a gives one example extracted from Figure 4.5 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 initialised 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 names have a single definition by placing extra phi-nodes where needed so that instructions can be renamed appropriately. Figure 4.10b shows how the property violation in Figure 4.10a can be corrected using this strategy.



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 4.11: Phi-node coalescing reduces the number of phi-nodes and selections.

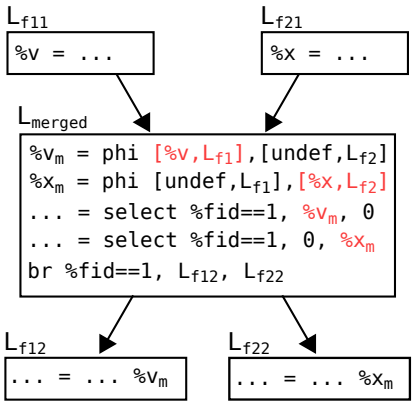
4.1.4 Phi-Node Coalescing

The approach described in Section 4.1.3 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 optimisation technique, *phi-node coalescing*, that SalSSA uses to lower register pressure.

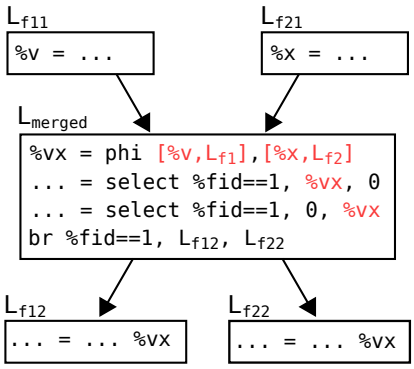
Figure 4.11 illustrates such an optimisation opportunity. SalSSA is merging an instruction with different arguments, so it needs to select the right one based on the function identifier. The two arguments though, `v` and `x`, have *disjoint definitions*, i.e. they have non-merged definitions from different input functions. Using the standard SSA construction algorithm would result in the sub-optimal code shown in Figure 4.11a. This code inserts two trivial phi-nodes to select, again, `v` or `x` based on the executed function. SalSSA optimises this code by coalescing both phi-nodes into a single one and removing the selection statement. As shown in Figure 4.11b, the optimised 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 4.12 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,



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 4.12: Reducing the number of phi-nodes by coalescing disjoint definitions with no user instructions in common.

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 optimisation pairs disjoint definitions that maximise 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 optimisation chooses pairs $(d_1, d_2) \in S_1 \times S_2$ that maximise 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.

4.2 Evaluation

Chapter 5

Optimisation Strategy

5.1 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, which is done with the help of the compiler's target-specific cost model.

The compiler's cost model provides a target-dependent cost estimation that approximates the cost of an intermediate representation (IR) instruction when lowered to machine instructions. Compilers usually provide a performance cost model, which estimates the latency of each instruction, and a code-size cost model, which estimates the binary size of each instruction. These cost models play an essential role in the decision making of most compiler optimisations [41, 48]. The absolute cost estimated for one instruction is less important than its relation to the cost of other instructions. Ultimately, these costs are used for comparing two pieces of code, informing the compiler's decision of whether or not a given transformation is profitable.

In order to estimate code-size cost of one function, we sum up the code-size cost of all instruction by querying the cost model. 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 maximise 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, these cost models are expected to contain inaccuracies. Because we are operating on the IR level, one IR instruction does not necessarily translate to one machine instruction. Several number of optimisations and code transformations will run afterwards, modifying the code. Moreover, we cannot know exactly how each IR instruction will be lowered without actually running the compiler's backend. The same IR instruction can be lowered to different machine instructions, depending on its surrounding context, the instruction selection algorithm, and many other factors. Therefore, there is also an inherent limitation of estimating the cost of each instruction separately of its context. However, the use of cost models is still a good trade-off between compilation time and accuracy.

5.2 Exhaustive Search

5.3 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 optimisation 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 maximises 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 5.1 for our optimisation.

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

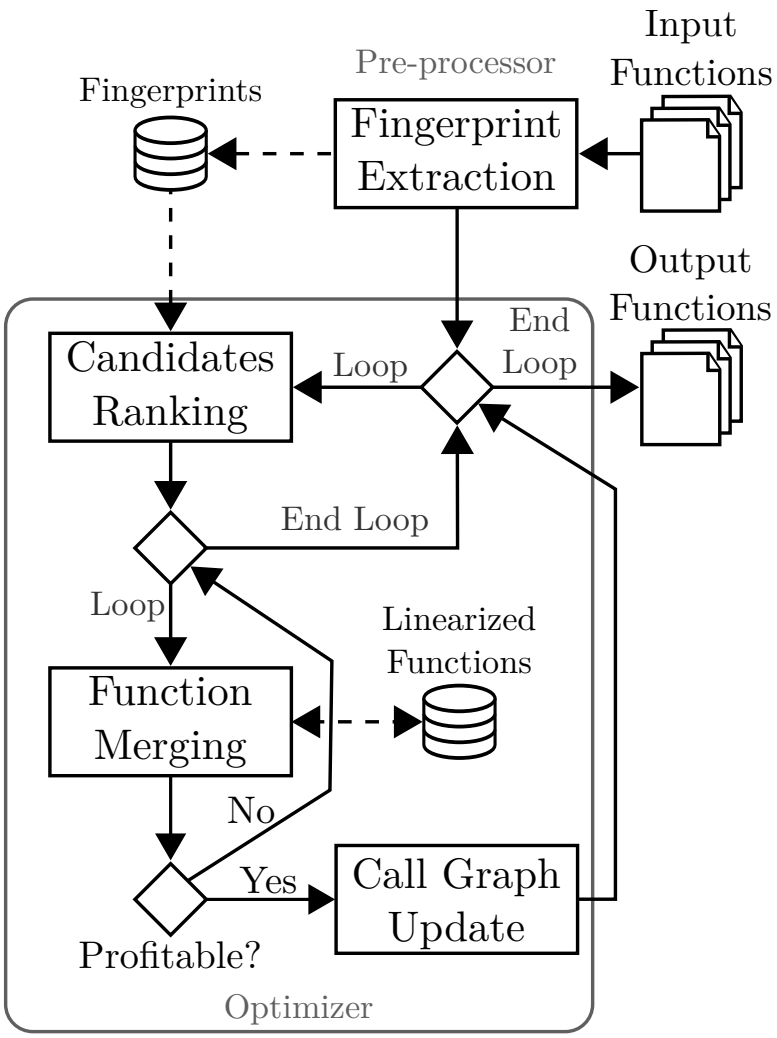


Figure 5.1: Overview of our exploration framework.

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

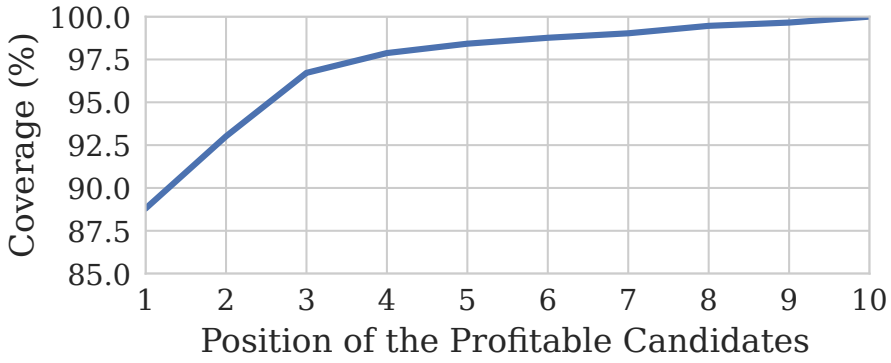


Figure 5.2: Average CDF for the position of the profitable candidate and the percentage of merged operations covered. 89% of the merge operations happen with the topmost candidate.

with the same data type could match.

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

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 5.2 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 optimisation working list. Because of this feedback loop, merge operations can also be performed on functions that resulted from previous merge operations.

5.3.1 Link-Time Optimisation

There are different ways of applying this optimisation, with different trade-offs. We can apply our optimisation 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 optimisation opportunities, since only pairs of functions within the same translation unit would be merged.

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

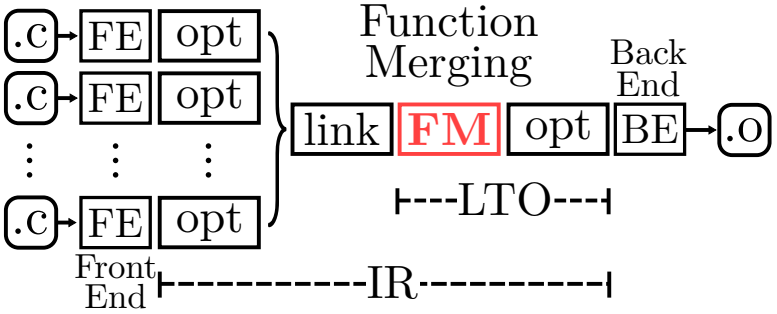


Figure 5.3: In our experiments we use a compilation pipeline with a monolithic link-time optimisation (LTO).

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 optimisation 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.

5.4 Evaluation

In this section, we evaluate the proposed optimization, where we analyze our improvements on code size reduction, as well as its impact on the program's performance and compilation-time.

5.4.1 Code-Size Reduction

Figure 5.4 reports the code size reduction over the baseline for the linked object. We observe similar trends of code size reduction on both target architectures. This is expected because the optimizations are applied at the platform-independent IR level. Changing the target architecture introduces only second order effects, such as slightly different decisions due to the different cost model (LLVM's TTI) and differences in how the IR is encoded into binary.

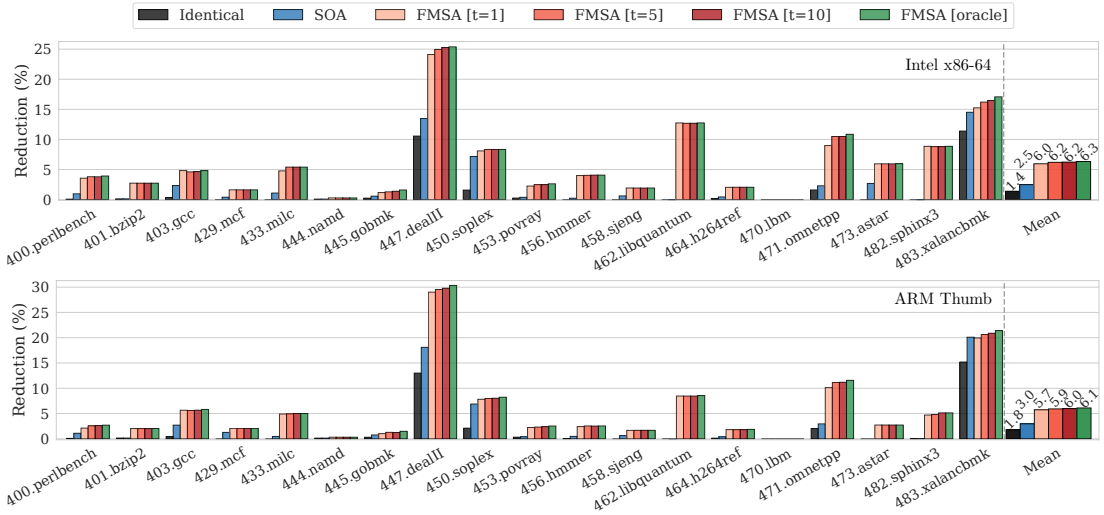


Figure 5.4: Object file size reduction for Intel (top) and ARM (bottom). We evaluate our approach (FMSA) under four different exploration thresholds, which control how many potential merging pairs we examine for each function before making a decision. Even for a threshold of one, we outperform the state-of-the-art by 2.4× (Intel) and 1.9× (ARM).

Our approach, FMSA, significantly improves over the state-of-the art (SOA). For the Intel platform, FMSA can achieve an average code size reduction of up to 6.3% (or 6% with the lowest exploration threshold), while the SOA and Identical had an average reduction of 2.5% and 1.4%, respectively. Similarly, for the ARM platform, FMSA can achieve an average code size reduction of up to 6.1% (or 5.7% with the lowest threshold), while SOA and Identical had an average reduction of 3% and 1.8%, respectively.

For several of the benchmarks, the proposed technique achieves impressive code size reduction compared to other merging approaches.

In most cases, LLVM's identical function merging has very little impact on code size. We see noticeable impact only on some of the C++ benchmarks, namely, `447.dealII`, `450.soplex`, `471.omnetpp`, `483.xalancbmk`. These are the cases that identical function merging was designed to handle, duplicate functions due to heavy use of templating. Although the state-of-the-art improves over LLVM's identical function merging, it still gets most of its code size reduction for benchmarks with heavy use of templating. In addition to achieving better results in all of these cases, our technique also shows remarkable reductions on several of the C benchmarks, especially `462.libquantum` and `482.sphinx3`, where other techniques have no real impact.

In Section **??**, we show two examples extracted from `462.libquantum` and `482.sphinx3`, where we detail how existing techniques fail to merge similar functions in these benchmarks. Our technique is the *first* that can handle these examples, producing merged functions equivalent to the handwritten ones shown in Figures **??** and **??**.

Table 5.1: Number and size of functions present in each SPEC CPU2006 benchmark just before function merging, as well as number of merge operations applied by each technique.

Benchmarks	#Fns	Min/Avg/Max Size	Identical	SOA	FMSA[t=1]	FMSA[t=10]
400.perlbench	1699	1 / 125 / 12501	12	103	175	200
401.bzip2	74	1 / 206 / 5997	0	0	7	7
403.gcc	4541	1 / 127.7 / 20688	136	341	614	710
429.mcf	24	18 / 87.25 / 297	0	1	1	1
433.milc	235	1 / 67.69 / 416	0	6	26	34
444.namd	99	1 / 570.64 / 1698	1	1	5	5
445.gobmk	2511	1 / 43.22 / 3140	183	485	436	605
447.dealII	7380	1 / 60.63 / 4856	1835	2785	2974	3315
450.soplex	1035	1 / 73.27 / 1719	27	125	156	163
453.povray	1585	1 / 98.05 / 5324	60	112	193	212
456.hmmer	487	1 / 99.98 / 1511	3	16	45	47
458.sjeng	134	1 / 145.06 / 1252	0	5	11	11
462.libquantum	95	1 / 56.64 / 626	0	1	9	9
464.h264ref	523	1 / 171.42 / 5445	3	22	50	52
470.lbm	17	6 / 123.41 / 680	0	0	0	0
471.omnetpp	1406	1 / 26.9 / 611	45	69	227	270
473.astar	101	1 / 67.11 / 584	0	2	4	4
482.sphinx3	326	1 / 80 / 924	2	6	24	26
483.xalancbmk	14191	1 / 38.58 / 3809	3057	4573	4342	4887

Table 5.1 provides detailed statistics for the SPEC CPU2006. We show how many

functions (#Fns) are present in the linked program just before the merging pass, as well as the average, minimum, and maximum size of these functions, in number of instructions, at this same point in the compilation pipeline. We also report how many pairwise merge operations are performed by each one of the function merging techniques. Note that in almost all cases FMSA performs significantly more merge operations than the other techniques. There are only two cases where FMSA with exploration threshold of one finds fewer profitable merges than the state-of-the-art. This is due to our aggressive pruning of the search space with our ranking mechanism. Simply increasing the threshold, e.g. to ten, allows FMSA to merge more functions. In any case, these extra merge operations of the state-of-the-art have little effect on the overall code size reduction. The state-of-the-art is more likely to fail to merge large functions and succeed with small ones, so even when merging more functions, it does not reduce code size as much as FMSA.

MiBench: Embedded Benchmark Suite

We have shown that our technique achieves good results when applied on the SPEC CPU suite. It reduces size not only on templated C++ benchmarks, like other techniques, but also on C benchmarks where merging opportunities should be almost non-existant. Here, we further explore how FMSA handles such cases by applying it on the MiBench suite, a collection of small C programs each one composed of a small number of functions.

Figure 5.5 shows the object file reduction for the MiBench programs on the Intel platform. Our best result is for the `rijndael` benchmark, which implements the well-known AES encryption. FMSA merges the two largest functions, namely, `encrypt` and `decrypt`. Inspecting the LLVM IR for the `rijndael` benchmark, we observe that the two functions contain a total of 2494 instructions, over 70% of the code. When we merge them by sequence alignment, we create a single function with only 1445 instruction, a 42% reduction in the number of IR instructions. This translates into a 20.6% reduction in the linked object file.

Table 5.2 provides more detailed statistics for MiBench. LLVM achieves very limited results, reducing `jpeg_c` by just 0.13%, `jpeg_d` by 0.1%, and `ghostscript` by 0.02%, while having no effect on `typeset`. This happens because all the functions merged by LLVM's identical technique are tiny functions relative to the overall size of the program. Most of these functions comprise of just a few IR instructions. For example, in the `typeset` benchmark, while it is able to merge a pair of functions, they

Table 5.2: 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	Identical	SOA	FMSA[t=1]	FMSA[t=10]
CRC32	4	8 / 24.75 / 39	0	0	0	0
FFT	7	7 / 49.86 / 144	0	0	0	0
adpcm_c	3	37 / 73 / 100	0	0	0	0
adpcm_d	3	37 / 73 / 100	0	0	0	0
basicmath	5	4 / 70.8 / 232	0	0	0	0
bitcount	19	4 / 22.26 / 63	0	1	3	3
blowfish_d	8	1 / 245.38 / 824	0	0	0	0
blowfish_e	8	1 / 245.38 / 824	0	0	0	0
jpeg_c	322	1 / 100.52 / 1269	2	6	8	11
jpeg_d	310	1 / 98.93 / 1269	3	6	10	10
dijkstra	6	2 / 33 / 89	0	0	0	0
ghostscript	3446	1 / 54.2 / 4218	53	53	234	250
gsm	69	1 / 97.06 / 737	0	3	8	8
ispell	84	1 / 105.51 / 1082	0	2	5	5
patricia	5	1 / 77 / 167	0	0	0	0
pgp	310	1 / 88.52 / 1845	0	1	10	10
qsort	2	11 / 50 / 89	0	0	0	0
rijndael	7	46 / 472.29 / 1247	0	0	1	1
rsynth	46	1 / 97.28 / 778	0	0	0	0
sha	7	12 / 53.29 / 150	0	0	0	0
stringsearch	10	3 / 47.9 / 99	0	0	1	1
susan	19	15 / 291.84 / 1212	0	0	1	1
typeset	362	1 / 354.47 / 12125	1	4	31	35

only have five instructions. For the same benchmark, FMSA performs several merge operations, one of them between two functions with over 500 instructions. Overall, the state-of-the-art does slightly better than LLVM's identical technique but even in its best case it cannot reduce code size more than 0.7%.

Because these embedded benchmarks are much smaller than those in the SPEC suite, trivially similar functions are much less frequent. This is why neither the state-of-the-art nor LLVM's identical function merging technique had any real effect on these benchmarks. Our technique can look beyond trivially similar functions which allowed it to achieve significant code size reduction on these real embedded benchmarks.

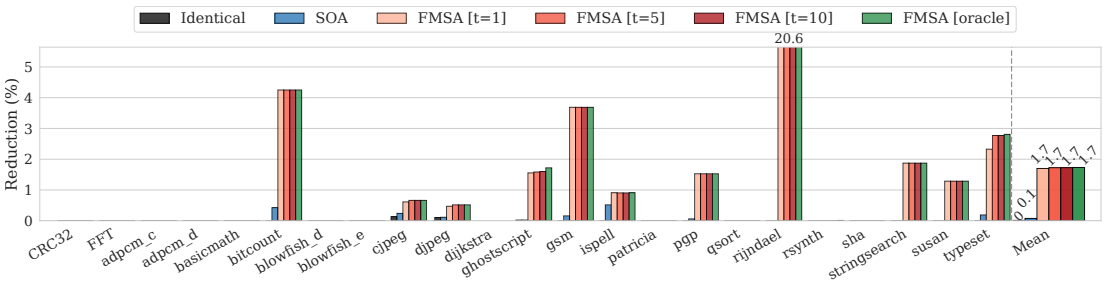


Figure 5.5: Object file size reduction for Intel on the Mibench benchmark suite. Our approach (FMSA) is the only one able to achieve a meaningful reduction on these benchmarks.

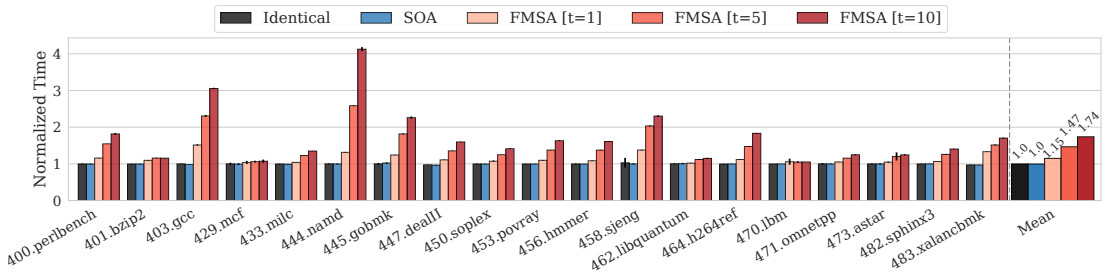


Figure 5.6: Compilation-time overhead on the Intel platform. For exhaustive exploration (not shown) the average overhead is 25×. Through ranking, we reduce overhead by orders of magnitude. For an exploration threshold of one, FMSA has an overhead of only 15%.

5.4.2 Compilation Overhead

Figure 5.6 shows the compilation-time overhead for all optimizations. As explained in the experimental setup, we only present results when compiling for the Intel platform. Since we cross-compile on the same machine for both targets, compilation times are very similar. We also do not include results for the oracle (exhaustive) exploration. It would be hard to visualize it in the same plot as the other configurations, since it can be up to 136× slower than the baseline.

Unlike the other evaluated techniques, our optimization is a prototype implementation, not yet tuned for compilation-time. We believe that compilation-time can be further reduced with some additional engineering effort. Nevertheless, by using our ranking infrastructure to target only the single most promising equivalent function for each function we examine, we reduce compilation-time overhead by up to two orders of magnitude compared to the oracle. This brings the average compile-time overhead to only 15% compared to the baseline, while still reducing code size almost as well

as the oracle. Depending on the acceptable trade-off between compilation-time overhead and code size, the developer can change the exploration threshold to exploit more opportunities for code reduction, or to accelerate compilation.

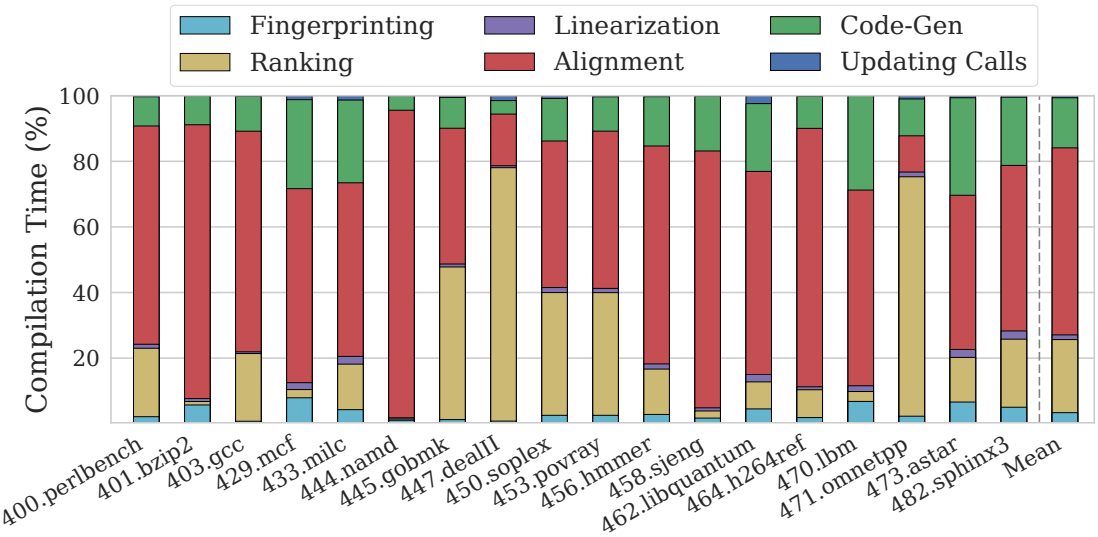


Figure 5.7: A compilation-time breakdown isolating the percentage for each major step of the optimization (t=1).

Figure 5.7 shows a detailed compilation-time breakdown. For each major step of the proposed optimization, we present the accumulated time spent across the whole program. To better understand the overhead of each step, we use an exploration threshold of one ($t = 1$). Because the ranking mechanism performs a quadratic operation on the number of functions, computing the similarity between all pairs of functions, it is expected that ranking would be amongst the most costly steps. However, it is interesting to notice that the sequence alignment dominates most of the compilation-time overhead, especially considering that this operation is performed only once per function, when $t = 1$. Although this operation is linear in the number of functions, the Needleman-Wunsh algorithm [39] is quadratic in the size of the functions being aligned, both in time and space. Unsurprisingly, code generation is the third most costly step, which also includes the time to optimize the merge of the parameters. The remaining steps contribute, in total, a small percentage of all the compilation-time overhead. This analysis suggests that optimizing the sequence alignment algorithm and the ranking mechanism is key to reducing even further the overall compilation-time overhead.

5.4.3 Performance Impact

The primary goal of function merging is to reduce code size. Nevertheless, it is also important to understand its impact on the programs' execution time and the trade-offs between performance and code size reduction. Figure 5.8 shows the normalized execution time. Overall, our optimization has an average impact of about 3% on programs' runtime. For most benchmarks, there is no statistically significant difference between the baseline and the optimized binary. Only for `433.milc`, `447.dealII`, and `464.h264ref` there is a noticeable performance impact.

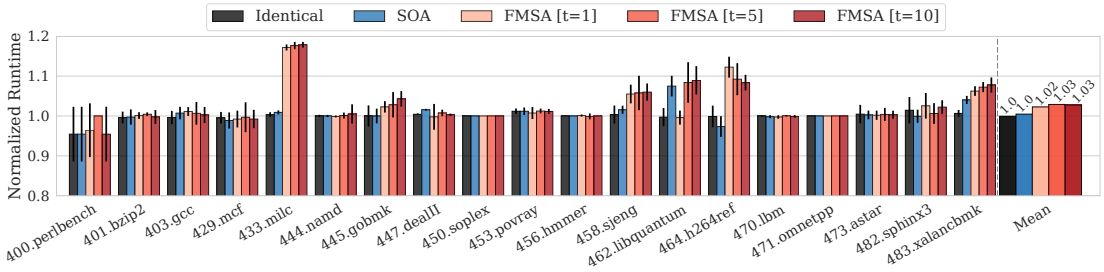


Figure 5.8: Runtime overhead on the Intel platform. Performance impact is almost always statistically insignificant. For the few benchmarks affected, FMSA merges hot functions.

We take `433.milc`, which has the worst result, for discussion. For an exploration threshold value of one, we merge 58 functions. Through profiling, we discovered that a handful of them contain hot code, that is, they have basic blocks that are frequently executed. If we prevent these hot functions from merging, all performance impact is removed while still reducing code size. Specifically, our original results showed a 5.11% code size reduction and an 18% performance overhead. Avoiding merging hot functions results in effectively non-existent performance impact and a code size reduction of 2.09%. This code size reduction is still about twice as good as the state-of-the-art. As with the compilation overhead, this is a trade-off that the developer can control.

Chapter 6

Reducing Runtime Overhead

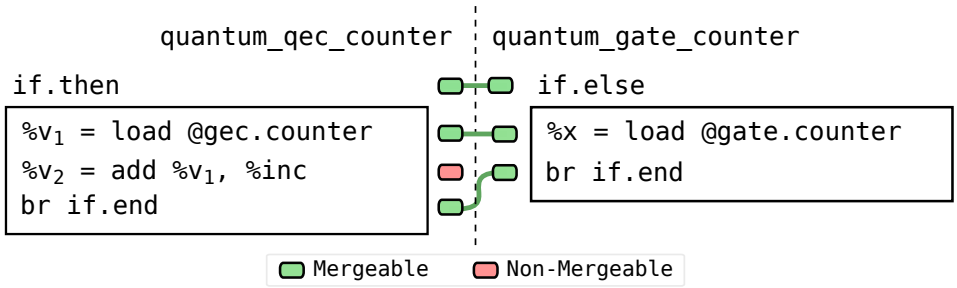
The identical function merging has no runtime penalty since it only alias two identical functions, removing one of the copies. However, merging partially similar functions may introduce runtime overheads in a given execution path.

Figure 6.1 shows an example of how function merging may introduce runtime overheads when merging two partially equivalent basic blocks. The two basic blocks involved in this merge operation were extracted from the `462.libquantum` benchmark. This example illustrates two different types of overheads that may be introduced during function merging: (i) the insertion of value selections; (ii) the insertion of branches to diverge to mismatching code or converge to matching code.

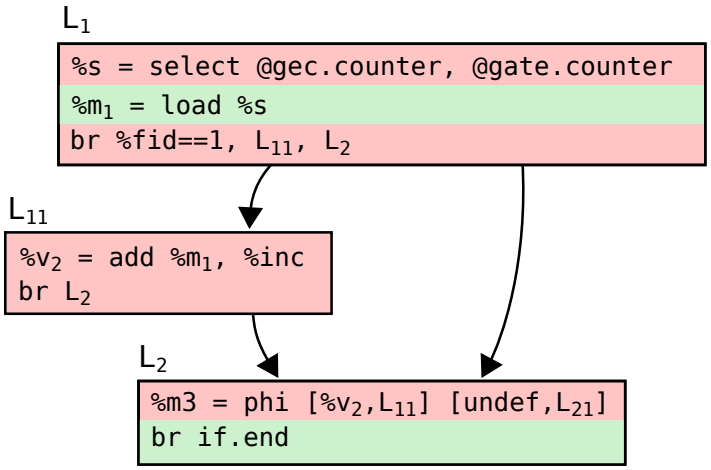
6.1 Conservative Function Merging

In this section, we propose a conservative function merging approach that aims at minimal runtime overhead in the absence of profiling information. The function merging approach described in Chapter 4 is capable of partially merging basic blocks, inserting branches between instructions to split matching from non-matching code. In order to minimise runtime overheads on possibly hot basic blocks, our conservative approach consider only merging whole basic blocks.

6.2 Profile-Guided Function Merging



(a) Pair of aligned basic blocks extracted from two larger functions in the `462.libquantum` benchmark.



(b) Merged code generated for the pair of aligned basic blocks.

Figure 6.1: Example of how function merging may introduce runtime overheads.

Chapter 7

Conclusion

Bibliography

[1] Android Oreo Go edition.

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

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

[4] Rein Aasland, Charles Abrams, Christophe Ampe, Linda J. Ball, Mark T. Bedford, Gianni Cesareni, Mario Gimona, James H. Hurley, Thomas Jarchau, VP Lehto, MA Lemmon, R Linding, BJ Mayer, M Nagai, M Sudol, U Walter, and SJ Winder. A one-letter notation for amino acid sequences. *European Journal of Biochemistry*, 5(2):151–153, 1968.

[5] T. M. Ahmed, W. Shang, and A. E. Hassan. An empirical study of the copy and paste behavior during development. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 99–110, 2015.

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

[7] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 159–170, New York, NY, USA, 1994. ACM.

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

[9] Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, 48(5):1073–1082, October 1988.

[10] John Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[11] Krzysztof Czarnecki. *Generative programming-principles and techniques of software engineering based on automated configuration and fragment-based component models*. PhD thesis, Verlag nicht ermittelbar, 1999.

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

[13] Dirk Draheim, Christof Lutteroth, and Gerald Weber. An analytical comparison of generative programming technologies. 2004.

[14] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen. Graph-based procedural abstraction. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 259–270, March 2007.

[15] Tobias J.K. Edler von Koch, Igor Böhm, and Björn Franke. Integrated instruction selection and register allocation for compact code generation exploiting freeform mixing of 16- and 32-bit instructions. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 180–189, New York, NY, USA, 2010. ACM.

[16] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 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, pages 85–94, New York, NY, USA, 2014. ACM.

[17] Sebastiana Etzo and Guy Collender. The mobile phone 'revolution' in Africa: Rhetoric or reality? *African Affairs*, 109(437):659–668, 2010.

[18] Joseph A Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[19] L. Ghica and N. Tapus. Optimized retargetable compiler for embedded processors - gcc vs llvm. In *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 103–108, 2015.

[20] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, 3(5):720–734, October 2016.

[21] Waqar Haque, Alex Aravind, and Bharath Reddy. Pairwise sequence alignment algorithms: A survey. In *Proceedings of the 2009 Conference on Information Science, Technology and Applications*, ISTA '09, page 96–103, New York, NY, USA, 2009. Association for Computing Machinery.

[22] Stuart L. Hart and Clayton M. Christensen. The great leap: Driving innovation from the base of the pyramid. *MIT Sloan Management Review*, 44(1):51–56, Fall 2002.

[23] Glenn Hickey and Mathieu Blanchette. A probabilistic model for sequence alignment with context-sensitive indels. In *Proceedings of the 15th Annual International Conference on Research in Computational Molecular Biology*, RECOMB'11, pages 85–103, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] Desmond G Higgins and Paul M Sharp. Fast and sensitive multiple sequence alignments on a microcomputer. *Bioinformatics*, 5(2):151–153, 1989.

[25] P. Jablonski and D. Hou. Aiding software maintenance with copy-and-paste clone-awareness. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 170–179, 2010.

[26] David A. Patterson John L. Hennessy. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann, 6 edition, 2017.

[27] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: Scalable and incremental lto. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 111–121. IEEE Press, 2017.

[28] S. L. Keoh, S. S. Kumar, and H. Tschofenig. Securing the internet of things: A standardization perspective. *IEEE Internet of Things Journal*, 1(3):265–275, June 2014.

[29] Miryung Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 83–92, 2004.

[30] J. Kolek, Z. Jovanović, N. Šljivić, and D. Narančić. Adding micromips backend to the llvm compiler infrastructure. In *2013 21st Telecommunications Forum Telfor (TELFOR)*, pages 1015–1018, 2013.

[31] Joseph B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

[32] 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.

[33] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint*, 2020.

[34] Christopher Lee, Catherine Grasso, and Mark F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 03 2002.

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

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

[37] David W. Mount. *Bioinformatics: Sequence and Genome Analysis*, volume 564. Cold Spring Harbor, NY: Cold Spring Harbor Laboratory Press, 2001.

[38] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

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

[40] Pedro Plaza, Elio Sancristobal, German Carro, Manuel Castro, and Elena Ruiz. Wireless development boards to connect the world. In Michael E. Auer and Danilo G. Zutin, editors, *Online Engineering & Internet of Things*, pages 19–27, Cham, 2018. Springer International Publishing.

[41] A. Pohl, B. Cosenza, and B. Juurlink. Cost modelling for vectorization on ARM. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 644–645, Sept 2018.

[42] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function merging by sequence alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 149–163, Piscataway, NJ, USA, 2019. IEEE Press.

[43] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. Compiling Java for low-end embedded systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 42–50, New York, NY, USA, 2003. ACM.

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

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

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

[47] 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.

[48] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. VW-SLP: Auto-vectorization with adaptive vector width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 12:1–12:15, New York, NY, USA, 2018. ACM.

[49] A. Varma and S. S. Bhattacharyya. Java-through-C compilation: an enabling technology for Java in embedded systems. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 3, pages 161–166 Vol.3, Feb 2004.

[50] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1(4):337–348, 1994.

[51] V. M. Weaver and S. A. McKee. Code density concerns for new architectures. In *2009 IEEE International Conference on Computer Design*, pages 459–464, Oct 2009.