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 [18, 24]. 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, 22].

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 [43]. 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 [16].

In such constrained scenarios, reducing the application footprint is essential [6, 30, 47, 48, 54]. 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 [19, 28, 46, 48]. Despite the importance of keeping code size small, production compilers offer little help beyond a few classical optimisations [8, 10, 13]. 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 [30, 56].

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, 13]. 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, 37]. 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, 34, 51]. 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, 37]. 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. [17] 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.

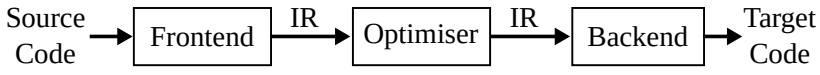


Figure 2.1: Overview of the three-phase compiler infrastructure.

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.

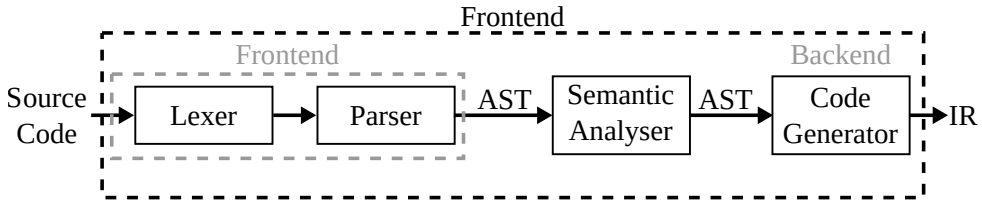
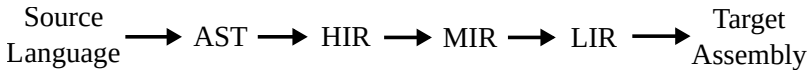
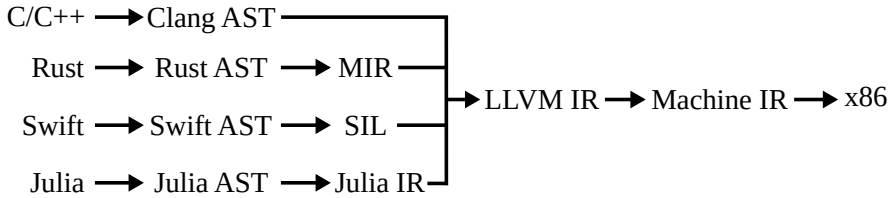


Figure 2.2: Breakdown of the frontend, illustrating how compilers are organised as a series of phases.



(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 [35]. 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.

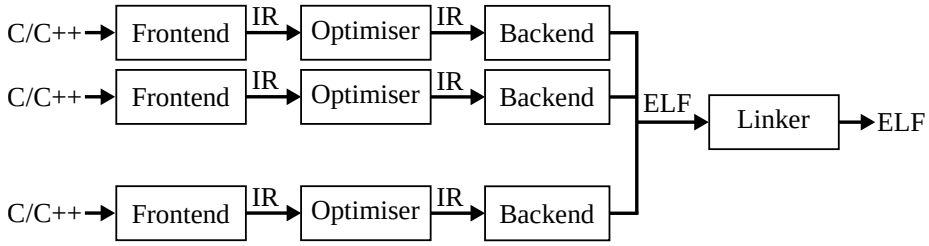


Figure 2.4: Overview of the three-phase compiler infrastructure.

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 [29].

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.

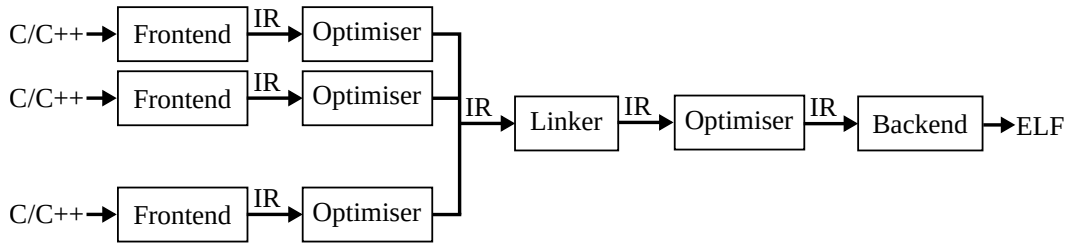


Figure 2.5: Overview of the three-phase compiler infrastructure.

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.

2.1.2 Target-Specific Cost Models

Compilers have cost models to compare two pieces of code in order to decide whether or not a given code transformation is profitable. Therefore, these cost models play an essential role in the decision making of most compiler optimisations [44, 53].

These cost models provide target-dependent cost estimations, with respect to a given metric, approximating the cost of an IR instruction when lowered to machine instructions. Compilers usually provide multiple cost models, including: a performance cost model that estimates the latency of each instruction; a code-size cost model that estimates the binary size of each instruction. However, these cost models need not predict precisely the latency or binary size of each instruction, because, Ultimately, these costs are used only for comparing two pieces of code in order to guide optimisations. Therefore, the cost estimated for one instruction is only relevant relative to other instructions.

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, 42, 49, 55] where it has been critical in the understanding of functional, structural, or evolutionary relationships between the sequences [33, 40].

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. [36], shows two protein sequences where amino acids are represented by their one-letter symbology [4].

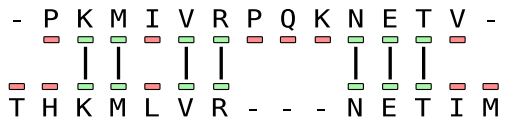


Figure 2.6: Example of an optimum alignment between two sequences. Matching segments are shown in green, vertically centred, and the non-matching segments are shown in red at the sides.

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., $S = (a_1, \dots a_k)$. Let $S_1, \dots, S_m$ be a set of sequences, possibly of different lengths but all derived from the same alphabet α, where $S_i = (a_1^{(i)}, \dots, a_{k_i}^{(i)})$, for all $i \in \{1, \dots, m\}$. Consider an extended alphabet that includes the *blank* character "$-$", i.e., $\beta = \alpha \cup \{-\}$. An alignment of the m sequences, $S_1, \dots, S_m$, is another set of sequences, $\bar{S}_1, \dots, \bar{S}_m$, 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 $\max\{k_1, \dots, k_m\} \leq l \leq k_1 + \dots + k_m$. Moreover, $\forall i \in \{1, \dots, m\}$, $\bar{S}_i = (b_1^{(i)}, \dots, b_l^{(i)})$, there are increasing functions $v_i : \{1, \dots, k_i\} \to \{1, \dots, l\}$, such that:

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

Finally, for all $j \in \{1, \dots, l\}$, 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 [23]. 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 [40]. 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 [40].

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, 25, 42, 49].

2.2.1 Needleman-Wunsch Algorithm

The Needleman-Wunsch algorithm [42] 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 [26].

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 2.7 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.

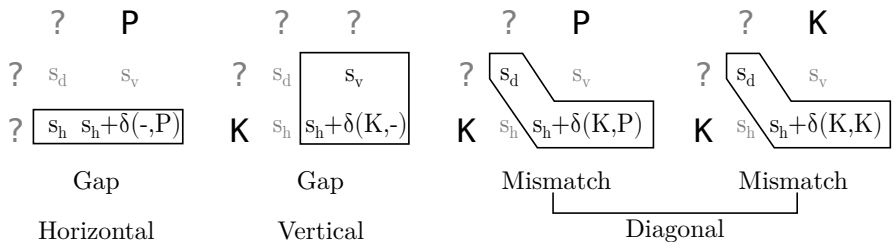
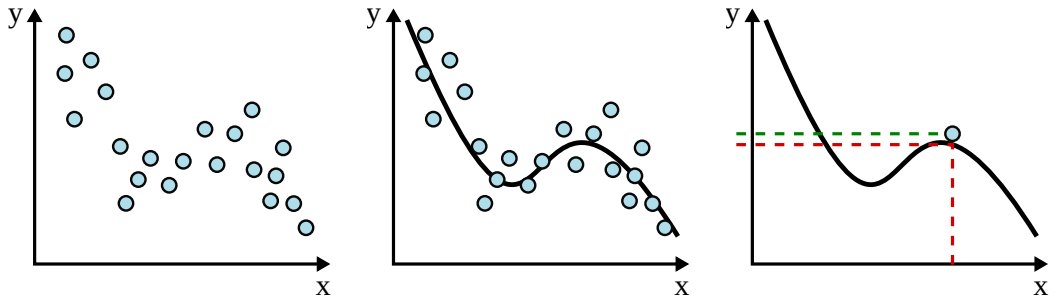


Figure 2.8: .

2.3 Machine Learning

Machine learning (ML) is the field of study of mathematical models that can automatically learn patterns from sample data, known as *training* data, in order to make predictions for unseen data. Regression and classification are two common types of machine learning application [21].

Regression concerns the prediction of a numerical value given some inputs, where an ML model learns the relationship between the outcome variable and one or more input features. Figure 2.9 illustrates an example of a regression model. Based on the sample data, a curve is fitted, minimising the overall error between the sample data and the curve. This curve represents the model that can be used to estimate the outcome variable for an unseen feature data.

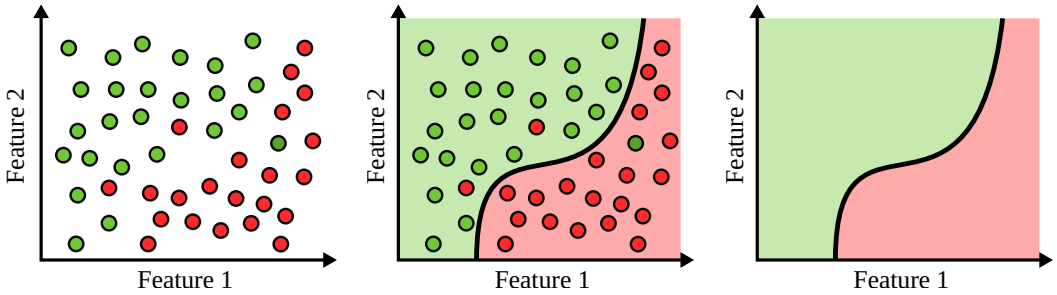


(a) Sample data. (b) Fitted curve. (c) Model.

Figure 2.9: An illustration of a regression model using machine learning. A regression model use the sample data to learn a function that maps the input features to an outcome numerical variable.

Classification concerns the prediction of a categorical label given some inputs, where an ML model learns the relationship between a finite set of labels and one or more input features. The training of a classifier requires inputs for which the category is known, called labelled training data. Figure 2.10 illustrates an example of a binary classification model. Based on the sample data, a curve is fitted, minimising the num-

ber of misclassification. This curve represents the model that can be used to predict the category of unseen feature data.



(a) Sample data. (b) Fitted curve. (c) Model.

Figure 2.10: An illustration of a classification model using machine learning. A regression model use the sample data to learn a function that maps the input features to an outcome numerical variable.

2.3.1 Neural Networks

Machine is commonly seen as a subset of artificial intelligence and a superset of deep-learning (DL), which is a family of machine learning methods based on artificial neural networks. Artificial neural networks (ANNs) are machine learning models that were vaguely inspired by the biological neural networks. These models are composed of artificial neurons, which are essentially mathematical functions known as *activation functions*, mapping inputs to a single output that can be fed to multiple other neurons. These neurons are grouped into layers and connected across layers by weighted edges. Figure 2.11 shows a simple feed-forward ANN with multiple layers of neurons, with fully connect layers, i.e., a neuron has an edge to every neuron in the next layer. The first layer is the input layer, with x_i being the input variables, and the last layer is the output layer, with y_i being the output variables.

2.3.1.1 Feed-Forward Neural Networks

A feed-forward neural networks is a simple but powerful neural network architecture where information flows through the layers in only one direction, namely, forward, as illustrated in Figure 2.11. This model defines a mapping $\mathbf{y} = f(\mathbf{x};\theta)$, were θ represents the parameters that are learnt, optimising the function approximation. The architecture of the feed-forward network is a direct acyclic graph, forming what is also known as

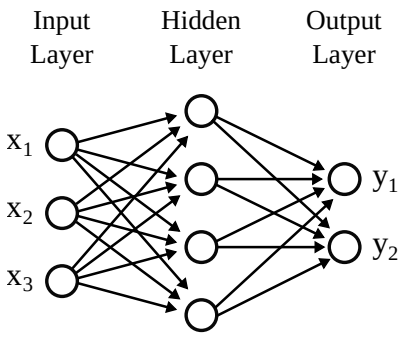


Figure 2.11: Example of an artificial neural network with multiple layers of neurons. Each circle represents an artificial neuron. Adjacent layers of neurons are fully connected.

a multi-layer perceptrons. The number of layers of the architecture is known as the depth of the model, with the final layer being the output layer.

2.3.1.2 Recurrent Neural Networks

2.3.1.2.1 Gated Recurrent Units

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, 13]. In this section, we will describe some of these classical size-reducing optimisations.

3.1.1 Constant Folding

Constant folding is an optimisation that operates on the instruction level, identifying instructions whose operands are constant values, performing the evaluation of the instruction at compile time, and replacing it by the resulting value. The effectiveness of constant folding can be augmented by combining it with constant propagation. Constant folding reduces code size by eliminating instructions that can be computed at compile time. Moreover, constant folding also works as an enabler to other optimisations, such as unreachable-code elimination (Section 3.1.2).

3.1.2 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 optimisations. For example, after constant propagation and constant folding, 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.

3.1.3 Dead-Code Elimination

A value definition is dead if it is not used on any path from the point in which it is defined to the exit point of the function. In a similar way, an instruction is dead if it computes only values that are not used on any execution path leading from the instruction. Any dead definition or instruction can be simply removed without altering the program's semantics, therefore reducing code size [41].

The algorithm to eliminate dead code has some similarities with that for unreachable-code elimination described in Section 3.1.2. This algorithm also works in a mark-sweep manner [52]. First, the algorithm marks *critical* instructions as *alive*. An instruction is *critical* if it has an observable effect, for example, if it is a return instruction, a branch instruction, a function call, a memory operation, any instruction with side effect, etc. Then, the algorithm follows the *use-def chain* of every alive instruction, marking the operand instructions as alive. This process continues until no more instructions can be marked as alive. Finally, the sweep phase removes all instructions that have not been marked as alive, reducing code size.

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, 27, 31]. 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 [20, 32, 41].

```
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.

```
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.

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 extract from the 447.dealII program in the SPEC CPU2006 [50] 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. Specialising `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* [34, 51], LLVM's *lld*, and the MSVC linker [2].

Before applying ICS, it is a common practice for linkers to place functions in separate sections [34, 51]. 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, 37]. 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, 27, 31]. The second pair of functions, shown in Figure 3.3b, are produced by generative programming [12, 14], where their code was automatically generated using a description language [20].

Von Edler et al. [17] 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.

```
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

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 5.1, 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 [38]. 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, 38**?**]. Procedural abstraction finds identical code that can be extracted into a separate function, replacing all replicated occurrences with a function call [15, 38].

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

Function Merging by Sequence Alignment

4.1 Motivation

In this section we make the argument for a more powerful function merging approach. Consider the examples from two SPEC CPU2006 benchmarks shown in Figures 4.1 and 4.2.

Figure 4.1 shows two functions from the `482.sphinx3` benchmark. The two functions are almost identical, only their function arguments are of different types, *float32* and *float64*, causing a single operation to be different. As shown at the bottom of Figure 4.1, these functions can be easily merged in three steps. First, we expand the function argument list to include the two parameters of different types. Then, we add a function identifier, `func_id`, to indicate which of the two functions is called. Finally, we place the lines that are unique to one of the functions in a conditional branch predicated by the `func_id`. Overall, merging these two functions reduces the total number of machine instructions by 18% in the final object file for the Intel x86 architecture.

Despite being so similar, neither GCC nor LLVM can merge the two functions. They can only handle identical functions, allowing only for type mismatches that can be removed by lossless bitcasting of the conflicting values. Similarly, the state-of-the-art [17], while more powerful, cannot merge the two functions either. It requires both functions to have the same list of parameters.

Figure 4.2 shows another two functions extracted from `462.libquantum`. While these two functions have the same signature, i.e. the same return type and list of parameters, they differ slightly in their bodies. Merging them manually is straightforward,

```
glist_t glist_add_float32(glist_t g, float32 val){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float32 = val;
  gn->next = g;
  return ((glist_t) gn);
}

glist_t glist_add_float64(glist_t g, float64 val){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  gn->data.float64 = val;
  gn->next = g;
  return ((glist_t) gn);
}
```

Merged Function

```
glist_t merged(bool func_id,
               glist_t g, float32 v32, float64 v64){
  gnode_t *gn;
  gn = (gnode_t *) mymalloc (sizeof(gnode_t));
  if (func_id)
    gn->data.float32 = v32;
  else
    gn->data.float64 = v64;
  gn->next = g;
  return ((glist_t) gn);
}
```

Figure 4.1: Example of two functions from the benchmark `sphinx` with different parameters that could be merged, as shown at the bottom. We highlight where they differ.

shown at the bottom of Figure 4.2, reducing the number of instructions by 23% in the final object file. But again, none of the existing techniques can merge the two functions. The state-of-the-art can work with non-identical functions, but it needs their CFGs to be identical. Even a single extra basic block, as in this case, makes merging impossible.

These examples show that all existing techniques are severely limited. Optimization passes in production compilers work only on effectively identical functions. State-of-the-art techniques can merge functions only when they are structurally identical, with isomorphic CFGs, and identical signatures. All of them miss massive opportunities for code size reduction. In the next sections, we show a better approach which removes such constraints and is able to merge arbitrary functions, when it is profitable to do so.

```
void quantum_cond_phase(
int control, int target, quantum_reg *reg){
  int i;
  COMPLEX_FLOAT z;
  if(quantum_objcode_put(COND_PHASE, control, target))
    return;
  z = quantum_cexp(pi / (1 << (control - target)));
  for(i=0; i<reg->size; i++) {
    if(reg->node[i].state & (1 << control)) {
      if(reg->node[i].state & (1 << target))
        reg->node[i].amplitude *= z;
    }
  }
  quantum_decohere(reg);
}
void quantum_cond_phase_inv(
int control, int target, quantum_reg *reg){
  int i;
  COMPLEX_FLOAT z;

  z = quantum_cexp(-pi / (1 << (control - target)));
  for(i=0; i<reg->size; i++) {
    if(reg->node[i].state & (1 << control)) {
      if(reg->node[i].state & (1 << target))
        reg->node[i].amplitude *= z;
    }
  }
  quantum_decohere(reg);
}
```

Merged Function

```
void merged(bool func_id,
int control, int target, quantum_reg *reg){
  int i;
  COMPLEX_FLOAT z;
  if(func_id)
    if(quantum_objcode_put(COND_PHASE, control, target))
      return;
  float var = (func_id)?pi:(-pi);
  z = quantum_cexp(var / (1 << (control - target)));
  for(i=0; i<reg->size; i++) {
    if(reg->node[i].state & (1 << control)) {
      if(reg->node[i].state & (1 << target))
        reg->node[i].amplitude *= z;
    }
  }
  quantum_decohere(reg);
}
```

Figure 4.2: Example of two functions from the benchmark `libquantum` with different CFGs that could be merged, as shown at the bottom. We highlight where they differ.

4.2 Our Approach

In this section we describe our proposed function merging technique and show how it merges the motivating examples. Our technique works on any two arbitrary functions, even when they have few similarities and merging them would be counter-productive. For that reason, we also introduce a cost model to decide when it is beneficial to merge two functions (see Section 4.3.1). To avoid an expensive quadratic exploration, we integrate our profitability analysis with an efficient ranking mechanism based on a lightweight fingerprint of the functions.

4.2.1 Overview

Intuitively, when we are manually merging two functions, in a textual format, we try to visualize them side by side, identifying the equivalent segments of code and the non-equivalent ones. Then, we use this understanding to create the merged function. In this paper, we propose a technique that follows this simple yet effective principle.

At the core of our technique lies a sequence alignment algorithm, which is responsible for arranging the code in segments that are either equivalent or non-equivalent. We implement this technique at the level of the intermediate representation (IR). Our current implementation assumes that the input functions have all their ϕ-functions demoted to memory operations, simplifying our code generation.

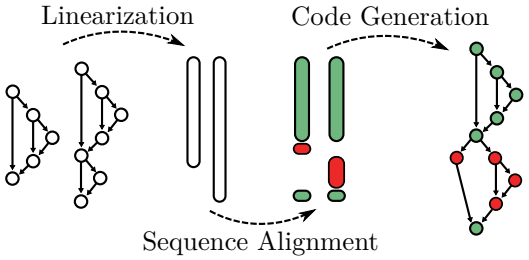


Figure 4.3: Overview of our function-merging technique. Equivalent segments of code is represented in light green and the non-equivalent ones in dark red.

The proposed technique consists of three major steps, as depicted in Figure 4.3. First, we linearize each function, representing the CFG as a sequence of labels and instructions. The second step consists of applying a sequence alignment algorithm, borrowed from bioinformatics, which identifies regions of similarity between sequences. The sequence alignment algorithm allows us to arrange two linearized functions into segments that are equivalent between the two functions and segments where they differ from one another. The final step performs the code generation, actually merging the two functions. Aligned segments with equivalent code are merged, avoiding redundancy, while the remaining segments where the two functions differ have their code guarded by a function identifier.

During code generation, we create a merged list of parameters, including the extra function identifier if there are any dissimilar segments. Arguments of the same type are shared between the functions, without necessarily keeping their original order. We also allow for the return types to be different, in which case we use an aggregate type to return values of both types. If one of them is void, then we do not create an aggregate type, we just return the non-void type. Given the appropriate function identifier, the merged function is semantically equivalent to the original functions, so we replace all of their invocations with the new function. It should be noted that in the special case where we merge identical functions, the output is also identical, emulating the behavior of function merging in production compilers.

After producing the merged function, the bodies of the original functions are re-

placed by a single call to this new function, creating what is sometimes called a *thunk*. In some cases, it may also be valid and profitable to completely delete the original functions, remapping all their original calls to the merged function. Two of the key facts that prohibit the complete removal of the original functions are the existence of indirect calls or the possibility of external linkage.

4.2.2 Linearization

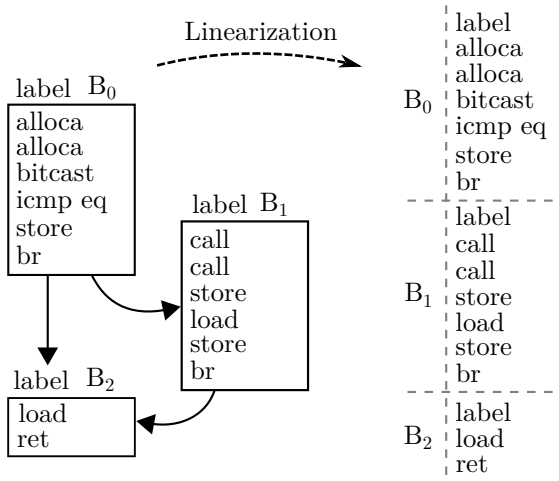


Figure 4.4: Linearizing the CFG of an example function.

Linearization[1] is a key step for enabling the use of sequence alignment. It takes the CFG of the function, specifies a traversal order of the basic blocks, and for each block outputs its label and its instructions. Linearization maintains the original ordering of the instructions inside each basic block. The edges of the CFG are implicitly represented with branch instructions having the target labels as operands. Figure 4.4 shows a simplified example of linearizing the CFG of a real function extracted from the SPEC CPU2006 `400.perlbench` benchmark.

The traversal order we use for linearization has no effect on the correctness of the transformation but it can impact its effectiveness. We empirically chose a reverse post-order traversal with a canonical ordering of successor basic blocks. This strategy leads to good performance in our experiments.

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

4.2.3 Sequence Alignment

When merging two functions, the goal is to identify which segments of the code are equivalent (and therefore can be merged) and which ones are different. To avoid breaking the semantics of the original program, we also need to maintain the order of the instructions for each of the functions.

After linearization, we reduce the problem of merging functions to the problem of *sequence alignment*. Sequence alignment is important in many scientific areas, most notably in molecular biology [9, 42, 49, 55] where it is used for identifying homologous subsequences of amino acid in proteins. Figure 4.5 shows an example of the sequence alignment between two linearized functions extracted from the `400.perlbench` benchmark, including the one used in Figure 4.4. 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 paired with blank characters.

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., $S = (a_1, \dots a_k)$. Let $S_1, \dots, S_m$ be a set of sequences, possibly of different lengths but all derived from the same alphabet α, where $S_i = (a_1^{(i)}, \dots, a_{k_1}^{(i)})$, for all $i \in \{1, \dots, m\}$. Consider an extended alphabet that includes the *blank* character "$-$", i.e., $\beta = \alpha \cup \{-\}$. An alignment of the m sequences, $S_1, \dots, S_m$, is another set of sequences, $\bar{S}_1, \dots, \bar{S}_m$, 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 $\max\{k_1, \dots, k_m\} \leq l \leq k_1 + \cdots + k_m$. Moreover, $\forall i \in \{1, \dots, m\}$, $\bar{S}_i = (b_1^{(i)}, \dots, b_l^{(i)})$, there are increasing functions $v_i : \{1, \dots, k_i\} \to \{1, \dots, l\}$, such that:

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

Finally, for all $j \in \{1, \dots, l\}$, 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.

Specifically for function-merging, we are concerned with the alphabet consisting of all possible typed instructions and labels. Every linearized function represents a

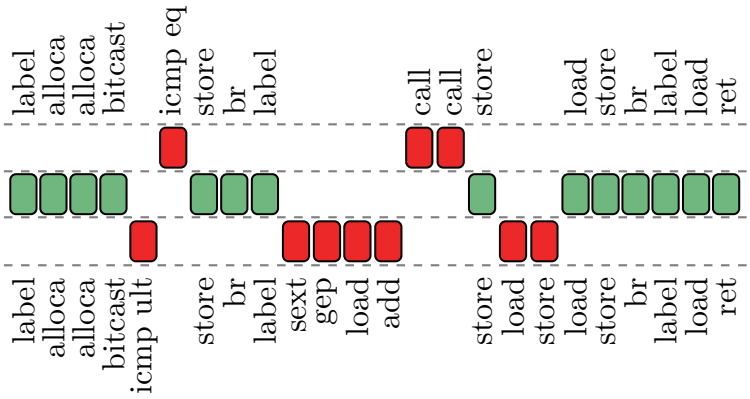


Figure 4.5: The sequence alignment between two functions, identifying the equivalent segments of code (green in the center) and the non-equivalent ones (red at the sides).

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.

Our work uses the Needleman-Wunsch algorithm [42] to perform sequence alignment. This algorithm gives an alignment that is guaranteed to be optimal for a given scoring scheme [26], however, other algorithms could also be used with different performance and memory usage trade-offs [9, 25, 42, 49]. Different alignments would produce different but valid merged functions.

The Needleman-Wunsch algorithm [42] is based on dynamic programming and consists of two main steps. First, it builds a *similarity matrix*, based on a scoring scheme, which assigns weights for matches, mismatches, and *gaps* (blank characters). Afterwards, a backward traversal is performed on the similarity matrix, in order to reconstruct the final alignment by maximizing the total score. We use a standard scoring scheme for the Needleman-Wunsch algorithm that rewards matches and equally penalizes mismatches and gaps.

4.2.4 Equivalence Evaluation

Before we merge functions, we first need to define what makes two pieces of code equivalent and therefore mergeable. In this section, we define equivalence in two separate cases, the equivalence between instructions and the equivalence between labels.

In general, two instructions are equivalent if: (1) their opcodes 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 equivalent if they can be

bitcast in a lossless way from one to the other. For pointers, we also need to make sure that there is no conflict regarding memory alignment. In the special case of function calls, type equivalence means that both instructions have identical function types, i.e. identical return types and identical list of parameters.

Labels can represent both normal basic blocks and landing blocks used in exception handling code. Labels of normal basic blocks are ignored during code equivalence evaluation, but we cannot do the same for landing blocks. We describe how we handle such blocks in more detail in the following section.

Exception Handling Code

Most modern compilers, including GCC and LLVM, implement the zero-cost Itanium ABI for exception handling [**?**] sometimes called the *landing-pad* model. This model consists of: (1) invoke instructions that have two successors, one for the normal execution and one for handling exceptions, called the landing block; (2) landing-pad instructions that encode which action is taken when an exception has been thrown. The invoke instruction co-operates tightly with its landing block. The landing block must have a landing-pad instruction as its first non-ϕ instruction. As a result, two equivalent invoke instructions must also have landing blocks with identical landing-pad instructions. This verification is made easy by having the landing-pad instruction as the first instruction in a landing block. Similarly, landing-pad instructions are equivalent if they have exactly the same type and also encode identical lists of exception and cleanup handlers.

4.2.5 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.

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 4.6 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.

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.

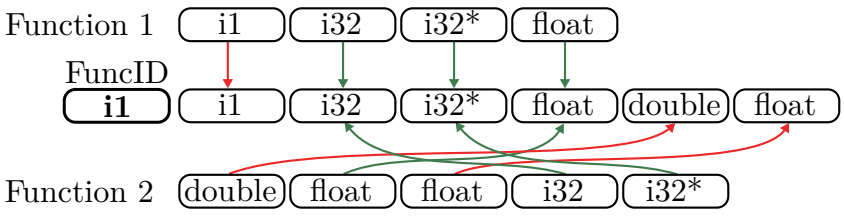


Figure 4.6: Example of a merge operation on the parameter lists of two functions.

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%.

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 in two passes over the aligned sequence. The first pass creates the basic blocks and instructions. The second assigns the correct operands to the instructions and connects the basic blocks. A two-passes approach is required in order to handle loops, due to cyclic data dependencies.

First, for each entry in the aligned sequence, we either create a new basic block for labels or we add a cloned instruction to the appropriate basic block. If the label represents a landing block, a landing-pad instruction is also added to the new basic block. During this process, we keep a mapping from the instructions and labels in the original functions to their corresponding values in the new merged function. We need this mapping to generate the use-definition chains for the merged function, which is done by pointing the operands of the instructions to the correct values in the function. However, at this point, the cloned instructions are given empty operands, as we are still creating the complete mapping.

While iterating over the aligned sequence, we also need to create extra basic blocks and branch instructions in order to maintain the semantics of the original functions, guarding the execution of instructions that are unique to one of the functions being merged. When transitioning from matching instructions or labels to non-matching ones, we need to branch to new basic blocks based on the function identifier. When transitioning back from non-matching segments to a matching segment, we need to reconnect both divergent points by branching back to a single new basic block where merged instructions will be added. This process generates diamond shaped structures in the CFG.

The second pass over the aligned sequence creates the operands of all instructions. We use the previously created mapping in order to identify the correct operands for each instruction in the merged function. There are two main cases: (1) Creating the operands for non-matching instructions (i.e. those that occur in just one function) is straightforward. In this case, we only need to use the values on which the operands of the original instruction map. (2) Matching instructions can have different values in corresponding operands in each one of the original functions. If this is the case and the original operands map to different values V_1 and V_2, then we need to choose at runtime the correct value based on the function identifier. We do with an extra select instruction "`select (func_id==1), V_1, V_2`", which computes the operand of the merged instruction. If the two values are statically identical, then we do not need a select.

If the operands are labels, instead of adding a select instruction, we perform operand selection through divergent control flow, using a new basic block and a conditional branch on the function identifier. If the two labels represent landing blocks, we hoist the landing-pad instruction to the new common basic block, converting it to a landing block and converting the two landing blocks to normal basic blocks. This is required for the correctness of the landing-pad model.

Similar to previous work on vectorization [45], we also exploit commutative instructions in order to maximize similarity. When assigning operands to commutative instructions, we perform operand reordering to maximize the number of matching operands and reduce the total number of select instructions required. It is also important to note that if we are merging two identical functions, no select or extra branch instruction will be added. As a result, we can remove the extra parameter that represents the function identifier.

4.3 Focusing on Profitable Functions

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

For every function, ideally, we would like to try to merge it with all other functions and choose the pair that maximizes the reduction in code size. However, this quadratic

exploration over all pairs of functions results in prohibitively expensive compilation overhead. In order to avoid the quadratic exploration of all possible merges, we propose the exploration framework shown in Figure 4.7 for our optimization.

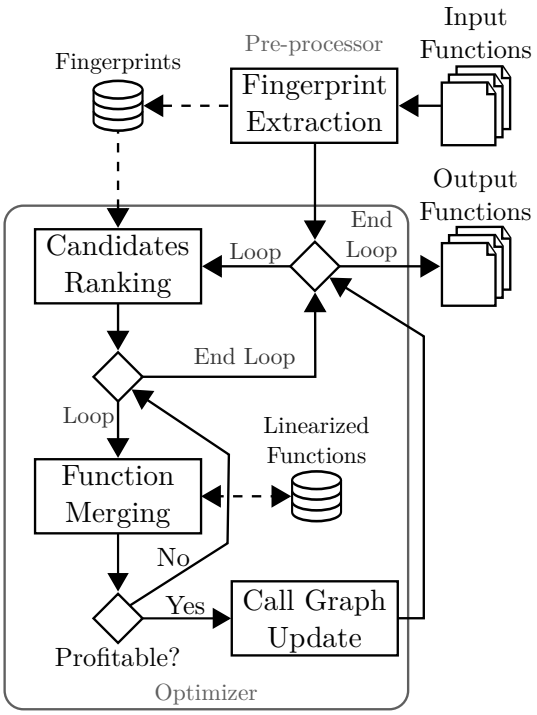


Figure 4.7: Overview of our exploration framework.

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

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

merge, this time based on type frequencies, which would happen if all instructions with the same data type could match.

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

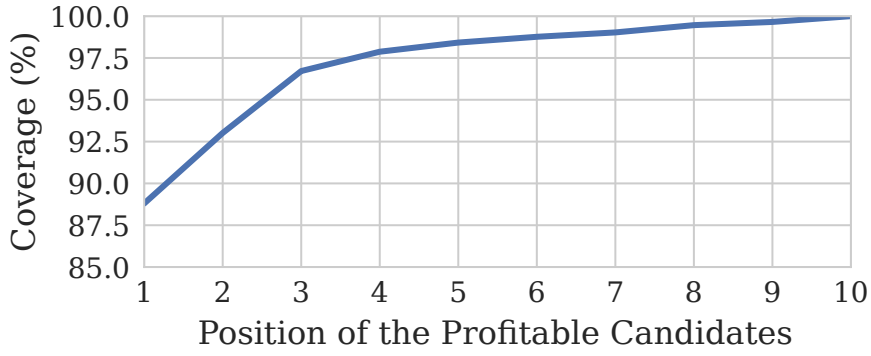


Figure 4.8: Average CDF for the position of the profitable candidate and the percentage of merged operations covered. 89% of the merge operations happen with the topmost candidate.

Ideally, profitable candidates will be as close to the top of the rank as possible. Figure 4.8 shows the cumulative distribution of the position of the profitable candidates

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

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

4.3.1 Profitability Cost Model

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

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

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

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

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

an IR instruction when lowered to machine instructions. Our implementation makes use of the code-size costs provided by LLVM's target-transformation interface (TTI), which is widely used in the decision making of most optimizations [44, 53].

4.3.2 Link-Time Optimization

There are different ways of applying this optimization, with different trade-offs. We can apply our optimization on a per compilation-unit basis, which usually results in lower compilation-time overheads because only a small part of the whole program is being considered at each moment. However, this also limits the optimization opportunities, since only pairs of functions within the same translation unit would be merged.

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

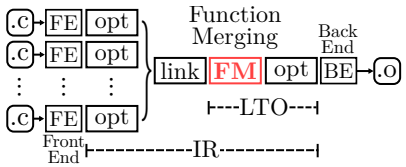


Figure 4.9: In our experiments we use a compilation pipeline with a monolithic link-time optimization (LTO).

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

Figure 4.9 shows an overview of the compilation pipeline used throughout our evaluation. First, we apply early code-size optimizations (`-Os`) to each compilation unit. Then, function merging and further code-size optimizations are applied during monolithic link-time optimization (LTO). With LTO, object file generation is delayed until all input modules are known, instead of being generated per translation unit, which enables more powerful optimizations based on whole-program analyses.

4.4 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.

4.4.1 Experimental Setup

We compare our optimization against the state-of-the-art [17] and LLVM's identical [3] function merging techniques. In our evaluation, we refer to the identical function merging as *Identical*, the state-of-the-art as *SOA*, and our approach as *FMSA*. We also run LLVM's identical function merging before both *SOA* and *FMSA*, as this helps to reduce compilation time by efficiently reducing the number of trivially mergeable functions.

All optimizations are implemented in LLVM v8 and evaluated on two benchmark suites: the C/C++ SPEC CPU2006 [50] and MiBench [**?**]. We target two different instruction sets, the Intel x86-64 and the ARM Thumb. Our Intel test bench has a quad-core 3.4 GHz Intel Core i7 CPU with 16 GiB of RAM. The ARM test bench has a Cortex-A53 ARMv8 CPU of 1.4 GHz with 1 GiB of RAM. We use the Intel platform for compiling for either target. As a result, compilation-time is almost identical for both targets. Changing the target only affects the behavior of the backend, a very short part of the pipeline. Because of that, we only report compilation-time overhead results for one of the targets, the Intel ISA.

For the proposed optimization, we vary the exploration threshold (Section 4.3) and we present the results for a range of threshold values. We also show the results for the oracle exploration strategy, which instead of using a rank-based greedy approach, merges a function with all candidates and chooses the best one. This oracle is a perfect ranking strategy but is unrealistic. It requires a very costly quadratic exploration, as explained in Section 4.3.

4.4.2 Code-Size Reduction

Figure 4.10 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.

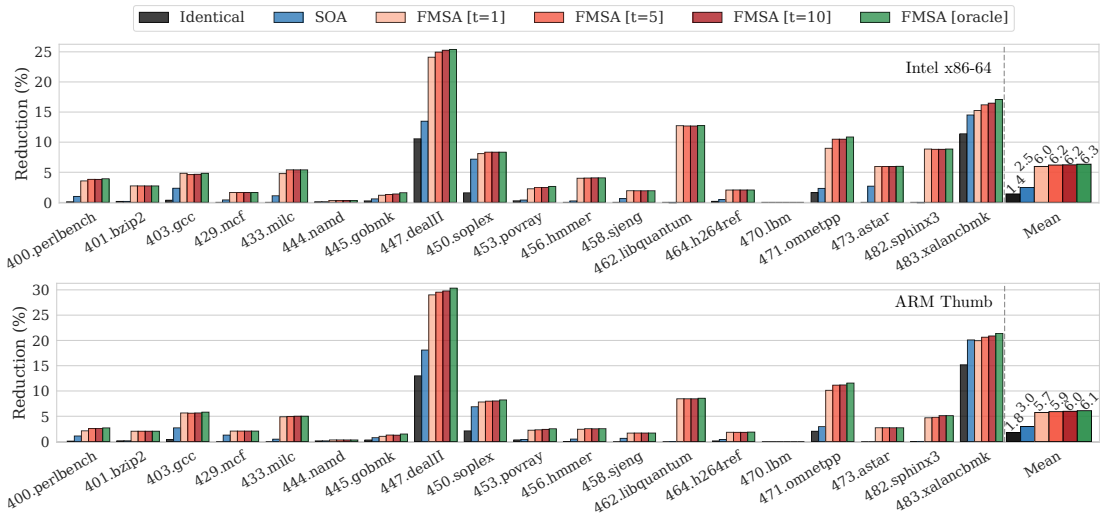


Figure 4.10: Object file size reduction for Intel (top) and ARM (bottom). We evaluate our approach (FMSA) under four different exploration thresholds, which control how many potential merging pairs we examine for each function before making a decision. Even for a threshold of one, we outperform the state-of-the-art 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 5.1, 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 4.1 and 4.2.

Table 4.1: 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 4.1 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 5.17 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.

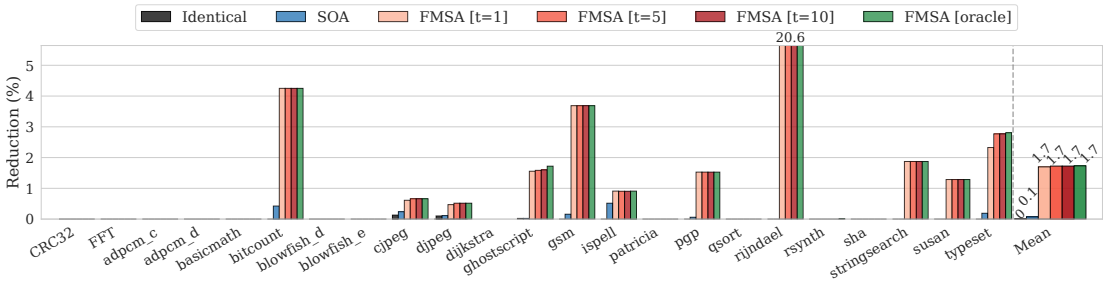


Figure 4.11: Object file size reduction for Intel on the Mibench benchmark suite. Our approach (FMSA) is the only one able to achieve a meaningful reduction on these benchmarks.

Table 5.1 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 only have five instructions. For the same benchmark, FMSA performs several merge operations, one of them between two functions with over 500 instructions. Overall,

Table 4.2: 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

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.

4.4.3 Compilation Overhead

Figure 5.23 shows the compilation-time overhead for all optimizations. As explained in the experimental setup, we only present results when compiling for the Intel plat-

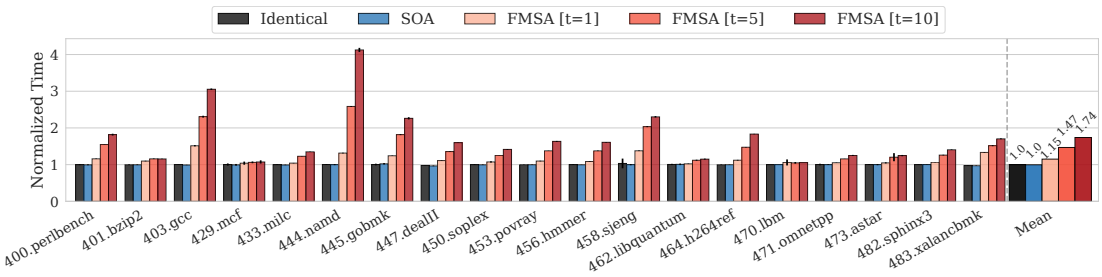


Figure 4.12: Compilation-time overhead on the Intel platform. For exhaustive exploration (not shown) the average overhead is 25×. Through ranking, we reduce overhead by orders of magnitude. For an exploration threshold of one, FMSA has an overhead of only 15%.

form. 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 4.13 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-Wunsch algorithm [42] is quadratic in the size of the functions being

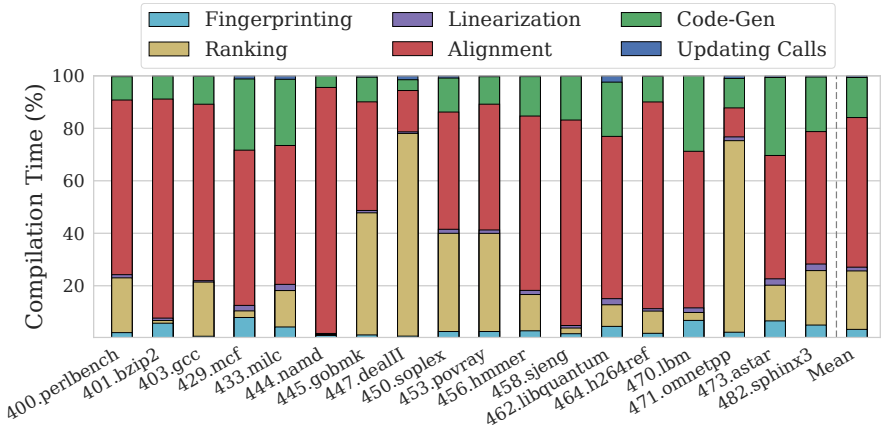


Figure 4.13: A compilation-time breakdown isolating the percentage for each major step of the optimization (t=1).

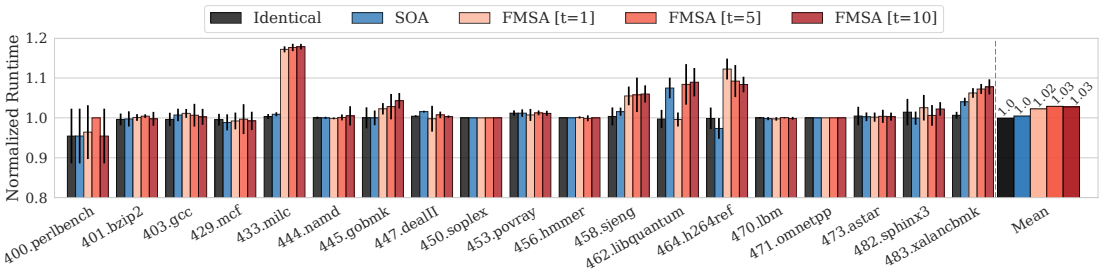


Figure 4.14: Runtime overhead on the Intel platform. Performance impact is almost always statistically insignificant. For the few benchmarks affected, FMSA merges hot functions.

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.

4.4.4 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 4.14 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.

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.

4.5 Conclusion

We introduced a novel technique, based on sequence alignment, for reducing code size by merging arbitrary functions. Our approach does not suffer from any of the major limitations of existing solutions, outperforming them by more than 2.4×. We also proposed a ranking-based exploration mechanism to focus the optimization on promising pairs of functions. Ranking reduces the compilation-time overhead by orders of magnitude compared to an exhaustive quadratic exploration. With this framework, our optimization is able to reduce code size by up to 25%, with an overall average of about 6%, while introducing an average compilation-time overhead of only 15%. Coupled with profiling information, our optimization introduces no statistically significant impact on performance.

For future work, we plan to focus on improving the ranking mechanism to reduce compilation time. We envisage further improvements can be achieved by integrating the function-merging optimization to a summary-based link-time optimization framework, such as ThinLTO in LLVM. We also plan to work on the linearization of the candidate functions, allowing instruction reordering to maximize the number of matches between the functions.

Chapter 5

Effective Function Merging in the SSA Form

5.1 Motivating Example

As a motivation example, consider the pair of input functions shown in Figure 5.1. While they are artificial, they highlight and isolate a problem that frequently appears in real programs, as we discuss later in Figure 5.4. These two functions have enough similarity to be profitably merged. A human expert could even replace them with the function shown in Figure 5.2, reducing the number of instructions by about 20%.

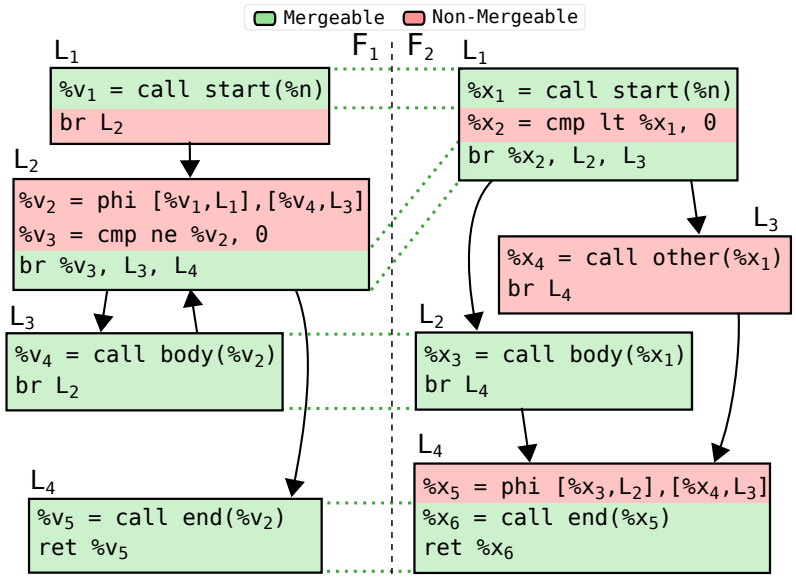


Figure 5.1: Original input functions to be merged, before register demotion. These simplified functions highlight a problem commonly seen in real programs.

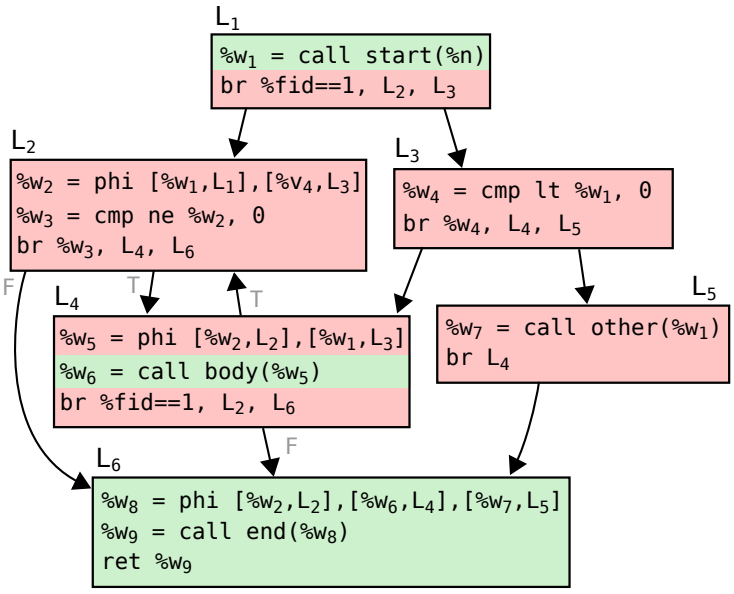


Figure 5.2: Desired merged function that can be produced by an expert. An extra argument called `%fid` is used to select between the two functions. This represents a gain of about 20% in the total number of instructions.

However, before aligning and automatically merging them, FMSA has to apply register demotion, as shown in Figure 5.3. *Phi-nodes* are removed and memory operations are created to propagate values across basic block boundaries. The sequence alignment algorithm then identifies the matching pairs of instructions (connected green marks), keeping the rest unaligned (in red).

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

In our example, we see in Figure 5.3 that some of the mergeable memory operations use different locations. One such case is the highlighted pair of *store* instructions. To maintain the semantics of the two functions after merging, the target address of the merged *store* will have to be selected based on the function identifier, either $\texttt{addr}_2$ or $\texttt{addr}_3$. Because the merged *store* instruction will not use the stack address directly, but instead a selected address, this prevents register promotion from eliminating these

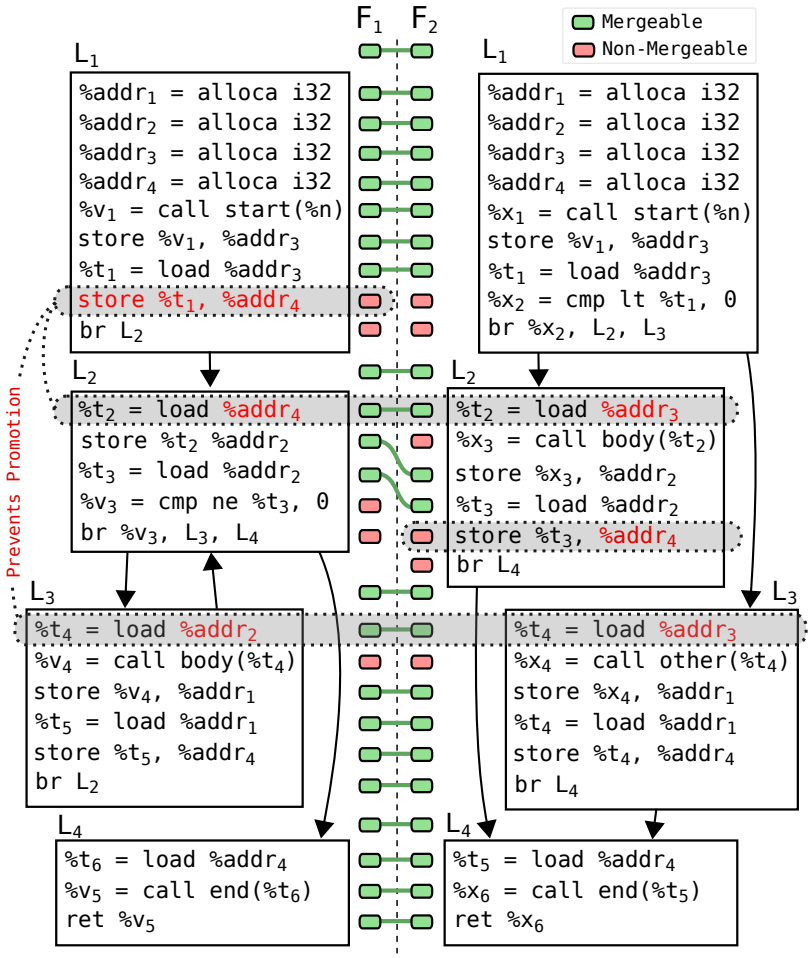


Figure 5.3: Aligned example functions after register demotion. The functions double in size after demotion, slowing down alignment. Merging some of the generated stack accesses will prevent eliminating them later through register promotion.

memory operations.

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

This kind of undesired scenario is likely to happen when merging two distinct functions after register demotion simply due to the sheer number of memory operations it creates. Figure 5.4 shows the average normalized size, before and after register demotion, across all functions in each program from the SPEC CPU2006 benchmark suite.

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

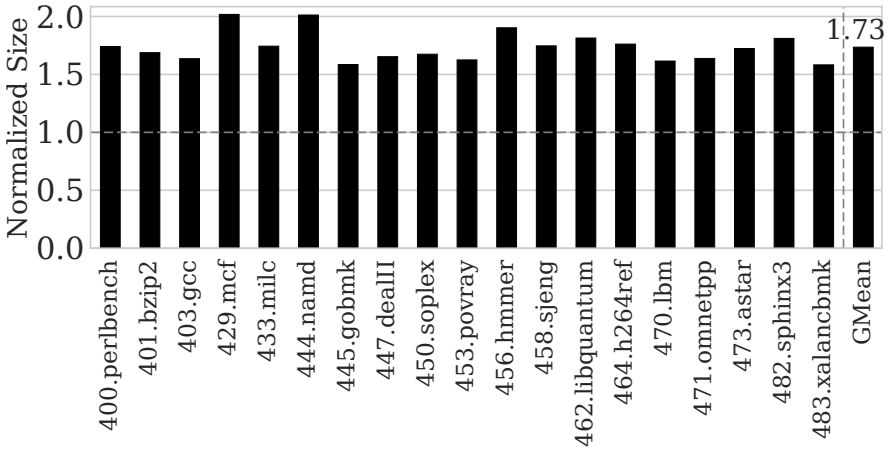


Figure 5.4: Average normalized function size, before and after register demotion, across all functions in each program from the SPEC 2006 benchmark suite. Register demotion increases function size by almost 75% on average.

Even for cases where the merge operation is profitable, register demotion remains a problem. Demotion artificially lengthens the functions to be aligned which in turns exacerbates the compile-time overheads associated with function merging. In our example, the combined size of the two input functions more than doubles, from 14 instructions in Figure 5.1 to 29 instructions in Figure 5.3. This increase is in line with what we have seen in SPEC CPU2006, including functions with many thousands of instructions. Regardless of whether register promotion will eventually remove the extra instructions or not, the alignment algorithm itself will have to process sequences twice as long. Since the memory usage and running time of the algorithm is quadratic in the sequence length, register demotion slows it down approximately by a factor of four. For applications with large functions after register demotion, the compile-time and memory usage overheads become prohibitive.

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

5.2 Our Approach

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

Road map In the rest of this section, we describe SalSSA, our novel approach for merging functions through sequence alignment with full support for the SSA form. By removing the need for preprocessing the input functions and performing register demoting, our approach is able to merge functions better and faster. Instead of translating the aligned functions directly to merged code, the SalSSA follows a top-down approach centered on the CFGs of the input functions. It iterates over the input CFGs, constructing the CFG of the merged function, interweaving matching and non-matching instructions (Section 5.2.1). Afterwards, all edges and operands are resolved, including appropriately assigning the incoming values to all *phi-nodes* (Section 5.2.2). SalSSA is designed to preserve all properties of SSA form via the standard SSA construction algorithm (Sections 5.2.3). Finally, SalSSA integrates a novel optimization with the SSA construction algorithm, called *phi-node coalescing*, producing even smaller merged functions (Section 5.2.4).

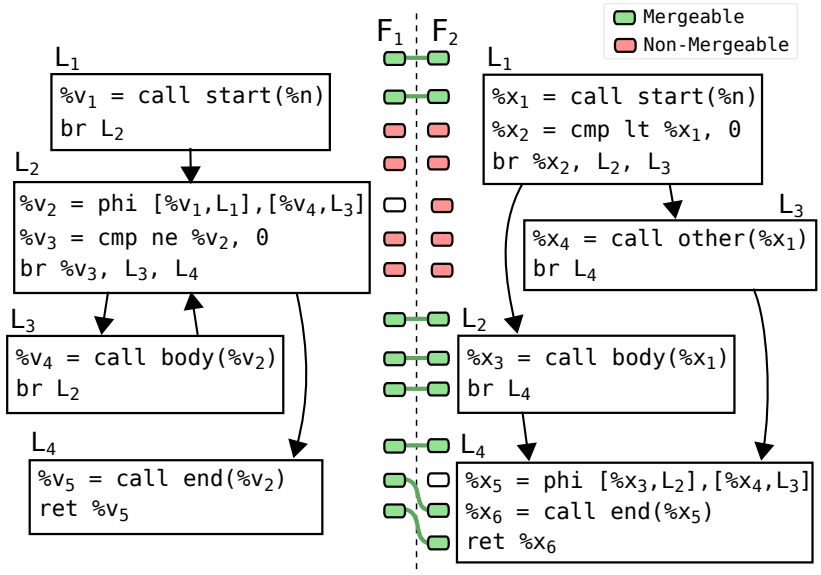


Figure 5.5: Example functions aligned without register demotion. *Phi-nodes* are excluded from alignment.

Working examples Figure 5.5 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.

5.2.1 Control-Flow Graph Generation

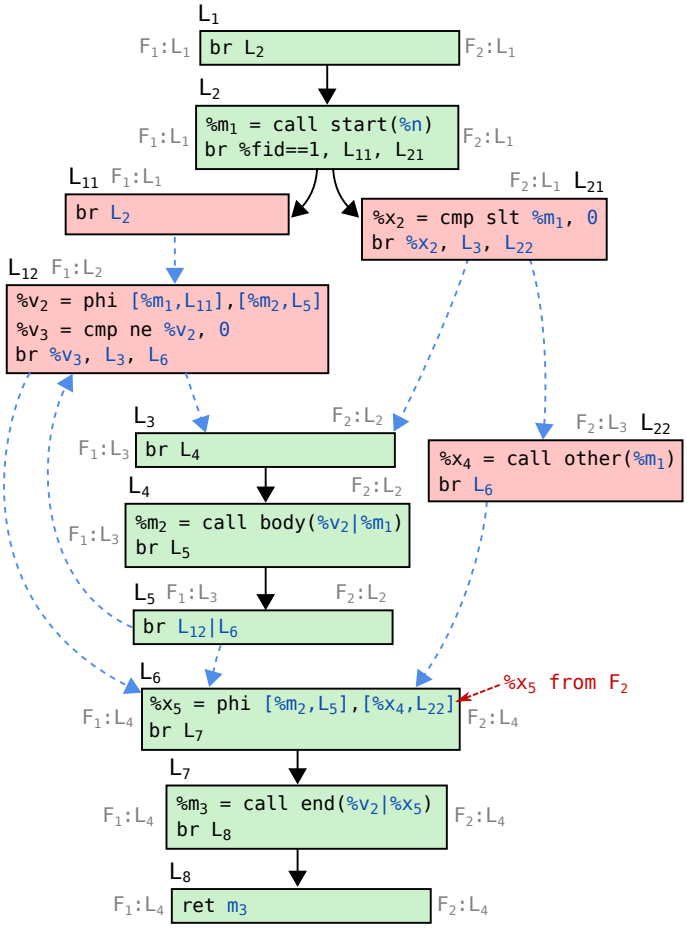


Figure 5.6: Merged CFG produced by SalSSA. Code corresponding to a single input basic block may be transformed into a chain of blocks, separating matching and non-matching code. The generator inserts conditional and unconditional branches to maintain the same order of instructions from the input basic block. Operands and edges highlighted in blue will be resolved by the operand assignment described in Section 5.2.2.

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 5.6 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 5.6, will be resolved later, during operand assignment.

5.2.1.1 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 5.6 shows an example where *phi-nodes* are present in basic blocks with both matching or non-matching labels. The phi-node $\texttt{x}_5$ is simply copied into the merged basic block labeled $\texttt{L}_6$.

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

5.2.1.2 Value Tracking

While generating the basic blocks and instructions for the merged function, SalSSA keeps track of two mappings that will be needed during operand assignment. The first

one, called *value mapping*, is responsible for 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 5.6. 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 5.2.2.3).

5.2.2 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 5.6). 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.

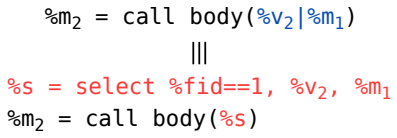


Figure 5.7: Operand selection for the `call` instruction in L_4 from Figure 5.6. Mismatching operands chosen with a `select` instruction on the function identifier.

Whenever the corresponding operands of merged instructions are different, we need a way to select the correct operand based on the function identifier. Section 5.2.2.1 describes how we perform label selection. In all other cases, we simply use a *select* instruction, as shown in Figure 5.7.

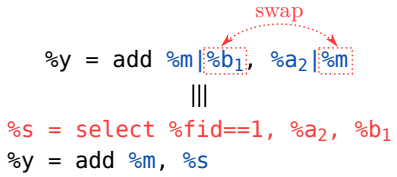


Figure 5.8: Optimizing operand assignment for commutative instructions. Example of a merged `add` instruction that can have its operands reordered to allow merging the two uses of `%m`, avoiding a *select* instruction.

When assigning operands to commutative instructions, we also perform operand

reordering to maximize the number of matching operands and reduce the need for *select* instructions. Figure 5.8 shows an example of a commutative instruction where an operand selection can be avoided by reordering operands. This property of commutative operations has been exploited before by other optimizations [45, 46**?**].

5.2.2.1 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 5.9 illustrates a CFG that handles label selection for a merged terminator instruction.

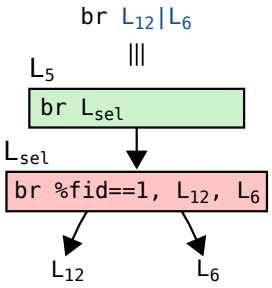
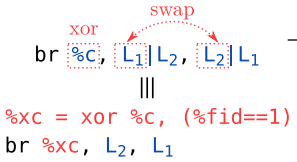


Figure 5.9: Label selection for mismatched terminator instruction operands L_{f1} and L_{f2} corresponding to labels of two different basic blocks. We handle control flow in a new basic block, L_{sel} with a conditional branch on the function identifier targeting the two labels. We use the label of the new block as the merged terminator operand.

Figure 5.10 shows a special case where we can also perform operand reordering on conditional branches that follow a specific pattern. When merging two conditional branches with matching label operands, except for their order, instead of creating two label selections, we can simply apply an *xor* operation on the condition and the function identifier, swaping the label operands for the true-value of the function identifier. As shown in Figure 5.10b, the xor operation flips the value of the condition for the true-value of the function identifier, preserving the semantic of the conditional branch.

This optimization adds the cost of one xor operation to avoid the cost of two label selections, which are implemented with branch instructions as shown in Figure 5.9.



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

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

(b) The truth table of the xor operation.

Figure 5.10: Optimizing label assignment for conditional branches. Example of a merged `br` instruction that can have its label operands reordered, trading two label selections by one xor operation.

5.2.2.2 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 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 5.11.

5.2.2.3 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

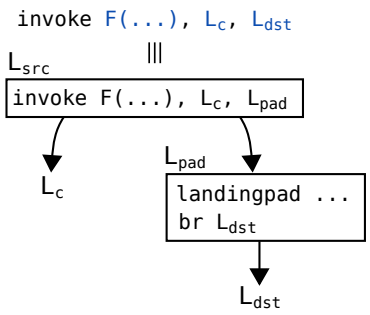


Figure 5.11: Landing blocks are added after operand assignment and are assigned to invoke instructions as operands.

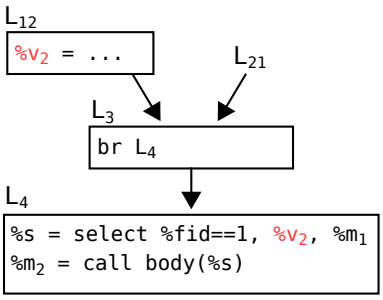
incoming flows present in the corresponding *phi-node* of their input function. For matching labels, each *phi-node* in the merged function will have additional incoming flows specific to the *other* input function but these flows should have undefined values.

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

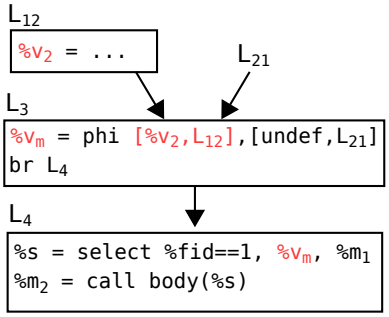
5.2.3 Preserving the Dominance Property

The code transformation process described so far could violate the *dominance property* of the SSA form. This property states that each use of a value must be dominated by its definition. For example, an instruction (or basic block) dominates another if and only if every path from the entry of the function to the latter goes through the former. Figure 5.12a gives one example extracted from Figure 5.6 where the dominance property is violated during code transformation. For this example, the dominance property is violated because $\%v_2$ is defined in block L_{12} and used in block L_4, but the former does not dominate the latter since there is a alternative path through L_{21}.

SalSSA is designed to preserve the dominance property to conform with the SSA form. It achieves this using a two-step approach. It first adds a *pseudo-definition* at the entry block of the function where names are defined and initialized with an *undefined* value. This guarantees that every register name will be defined on basic blocks from both functions. Then, SalSSA applies the standard SSA construction algorithm [**? ?**], which guarantees both the dominance and the single-reaching definition properties of



(a) Example where the dominance property is violated.



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

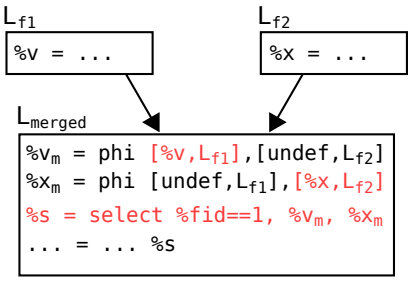
Figure 5.12: Example of how SalSSA uses the standard SSA construction algorithm to guarantee the dominance property of the SSA form.

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 5.12b shows how the property violation in Figure 5.12a can be corrected using this strategy.

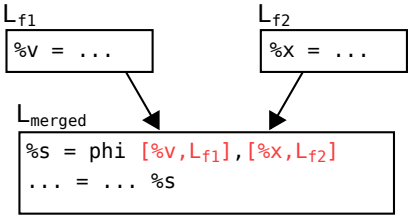
5.2.4 Phi-Node Coalescing

The approach described in Section 5.2.3 guarantees the correctness of the SSA form but generates extra phi-nodes and registers which increase register pressure and might lead to more *spill code*. In this section, we describe a novel optimization technique, *phi-node coalescing*, that SalSSA uses to lower register pressure.

Figure 5.13 illustrates such an optimization opportunity. SalSSA is merging an instruction with different arguments, so it needs to select the right one based on the function identifier. The two arguments though, `v` and `x`, have *disjoint definitions*, i.e. they have non-merged definitions from different input functions. Using the standard SSA



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

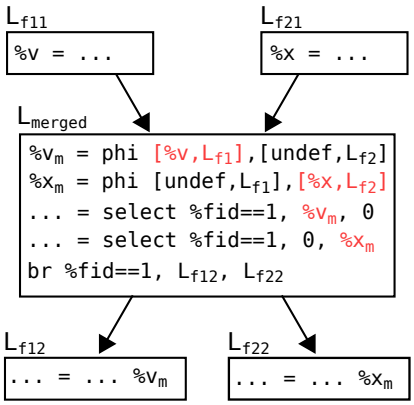
Figure 5.13: Phi-node coalescing reduces the number of phi-nodes and selections.

construction algorithm would result in the sub-optimal code shown in Figure 5.13a. This code inserts two trivial phi-nodes to select, again, `v` or `x` based on the executed function. SalSSA optimizes this code by coalescing both phi-nodes into a single one and removing the selection statement. As shown in Figure 5.13b, the optimized version has a smaller number of instructions and phi-nodes.

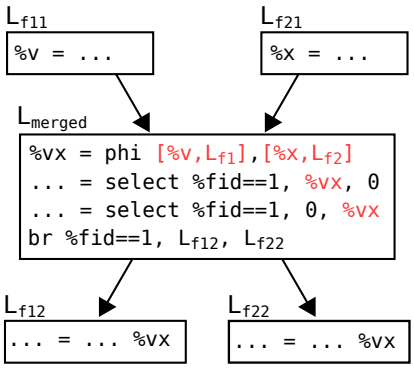
This transformation is valid because a value definition that is exclusive to a function will never be used when executing the other function. Figure 5.14 shows another example illustrating that even disjoint definitions that have no user instructions in common can be coalesced, reducing the number of phi-nodes.

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

Formally, the heuristic implemented in our phi-node coalescing can be described as follows. Given a set $S_1 \times S_2$ of disjoint definitions that violate the dominance property, the optimization chooses pairs $(d_1, d_2) \in S_1 \times S_2$ that maximize the intersection



(a) Phi-node placement without coalescing.



(b) Phi-node placement with coalescing.

Figure 5.14: Reducing the number of phi-nodes by coalescing disjoint definitions with no user instructions in common.

$UB(d_1) \cap UB(d_2)$, where $UB(d)$ is the set $\{Block(u) : u \in Users(d)\}$.

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

5.3 Evaluation

In this section, we compare SalSSA against the state-of-the-art algorithm of function merging by sequence alignment, FMSA [46]. We first present the code size reduction on the final object file. We then evaluate the compilation overhead and impact on program performance.

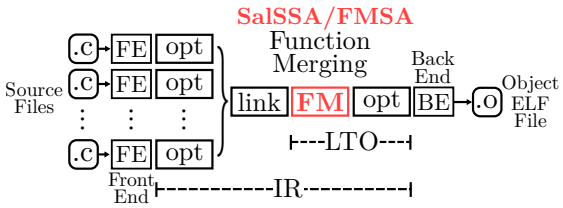


Figure 5.15: Compilation pipeline used for the evaluation. Both SalSSA and FMSA are applied in LTO mode.

5.3.1 Experimental Setup

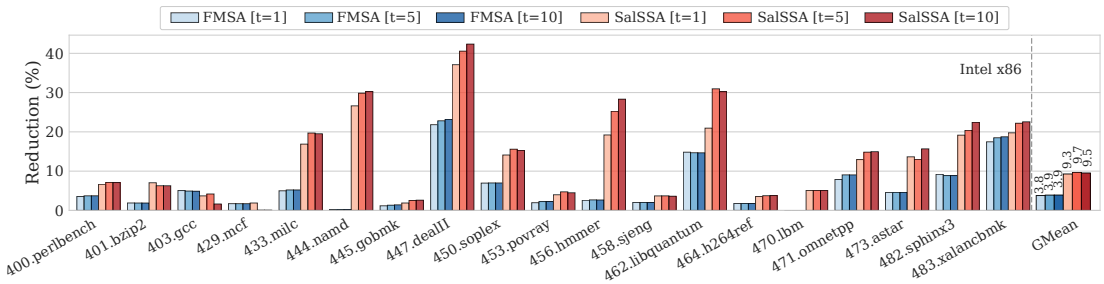
Most of our experiments directly compare SalSSA against FMSA [46]. We use the same compilation pipeline as our prior work [46], depicted in Figure 5.15. Both function merging optimizations are implemented in LLVM version 11.

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

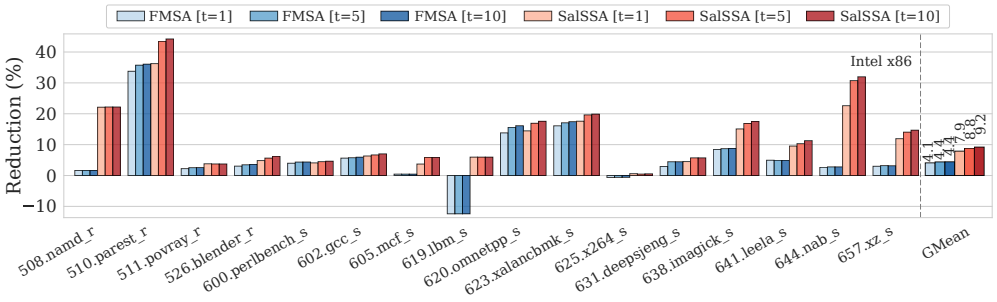
We evaluated SalSSA and FMSA on all C/C++ benchmarks of the SPEC CPU benchmark suite [50], both the 2006 and 2017 versions, targeting the Intel x86 architecture, and on the MiBench embedded benchmark suite targeting the ARM Thumb architecture. We run all experiments on a dedicated server with a quad-core Intel Xeon CPU E5-2650, 64 GiB of RAM, running Ubuntu 18.04.3 LTS. To minimize the effect of measurement noise, compilation and runtime overhead experiments were repeated 5 times.

5.3.2 Evaluation on SPEC CPU

Figures 5.16 reports the code size reduction on linked objects over the LLVM link-time optimizer (LTO). SalSSA significantly improves FMSA. With the lowest exploration threshold, SalSSA on average reduces the compiled code size by 9.3% and 7.9% on SPEC CPU2006 and CPU2017 respectively. These translate to nearly twice or above as much as FMSA, which achieves a 3.8% and 4.1% reduction on SPEC CPU2006 and CPU2017 respectively. The highest reductions are seen for `447.dealII` and `510.parset_r`, over 40% reduction. They are mainly due to the heavy use of tem-



(a) Results on SPEC CPU 2006.



(b) Results on SPEC CPU 2017.

Figure 5.16: Linked object size reduction over LLVM LTO when performing function merging with SalSSA or FMSA on SPEC CPU 2006 (a) and 2007 (b). Each approach was evaluated using three different exploration thresholds. On SPEC CPU2006, SalSSA reduces code size by 9.3% to 9.7% on average, almost twice as much as FMSA. On SPEC CPU2017, SalSSA reduces code size by 7.9% to 9.2% on average, more than twice as much as FMSA.

plate functions which leads to multiple similar functions. Other C++ programs display similar behavior, where SalSSA also achieves good code size reduction. SalSSA also gives remarkable code size reduction in many C programs, such as `456.hmmer`, `462.libquantum`, and `482.sphinx3`.

SalSSA outperforms FMSA for multiple benchmarks. The more pronounced cases are for `444.namd`, `456.hmmer`, `462.libquantum`, `447.dealII`, and `482.sphinx3` from SPEC CPU2006, as well as `508.namd_r`, `619.lbm_s`, `644.leela_s` and `657.xz_s` from SPEC CPU2017. These benchmarks were heavily affected by register demotion, as illustrated in Figure 5.4 for SPEC CPU2006. Similar to our motivating example in Section 5.1, when two non-identical functions have stack operations for nearly half of their instructions, misalignments become likely; these misalignments prohibit eliminating the merged stack operation through register promotion. This issue reduces the profit gained by FMSA. In some cases, like `619.lbm_s` and `625.x264_s`, the profitability cost model can fail, resulting in sufficient false positives to cause code bloating. We

will discuss this further in the next section.

5.3.3 Evaluation on MiBench

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

The MiBench suite is a collection of short C programs, each one composed of a small number of functions. When optimizing programs with a small number of functions, function merging optimizations will have fewer opportunities to find pairs of profitably merged functions. For example, the `qsort` program in MiBench has only two functions; as a result, neither FMSA nor SalSSA is able to merge them. As shown in Table 5.1, the same happens for other programs in the MiBench suite.

Figure 5.17 shows that SalSSA improves significantly over FMSA, achieving a geo-mean reduction of 1.4% to 1.6%, about twice as much as FMSA. This improvement comes from SalSSA's capability of generating better-merged functions, which leads to a larger number of profitable merge operations, as confirmed by Table 5.1.

Because FMSA requires register demotion to be applied to all functions before it can even attempt to merge them, FMSA ends up changing all functions even if no profitable merge operation is found. Figure 5.17 shows the effect of this preprocessing phase (denoted as *FMSA Residue*), which is obtained by running FMSA but not committing any merge operation. This FMSA Residue is the reason why FMSA sometimes has a non-zero code-size reduction (e.g., `adpcm_c`, `FFT`, `patricia`) despite not merging any functions. Since FMSA Residue might have an impact on the heuristics of later optimizations and code generation, its impact is almost random, sometimes being positive or negative on code-size. The impact of FMSA Residue is more noticeable in small programs, such as those found in MiBench, while in SPEC2006 it increases code size by only 0.02%, on average. To fix the issue highlighted by FMSA Residue, we would need to add an extra bookkeeping step of cloning all original functions so that we can rollback if they are not profitably merged. Fixing that would only increase unnecessarily the optimization complexity, but SalSSA offers a better solution where only merged functions are affected.

An interesting case is observed with both `cjpeg` and `djpeg`. Although SalSSA, with exploration threshold $t = 1$, increases code size, it is merging a superset of the

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

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

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

Figure 5.18 shows a breakdown for each merge operation performed by SalSSA, with exploration threshold $t = 1$, on the `djpeg` benchmark. We measured the impact of each merge operation, in isolation, to the size of the final object file. Although each

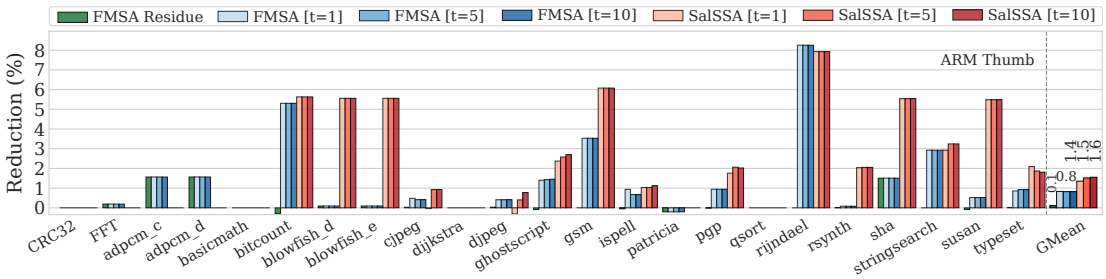


Figure 5.17: The percentual reduction in size of the linked object files, targeting the ARM architecture. We evaluate SalSSA or FMSA over the LLVM LTO on the MiBench embedded benchmark suite. Each approach was evaluated using three different exploration thresholds. SalSSA achieves a geo-mean reduction of 1.4% to 1.6%, about twice as much as FMSA.

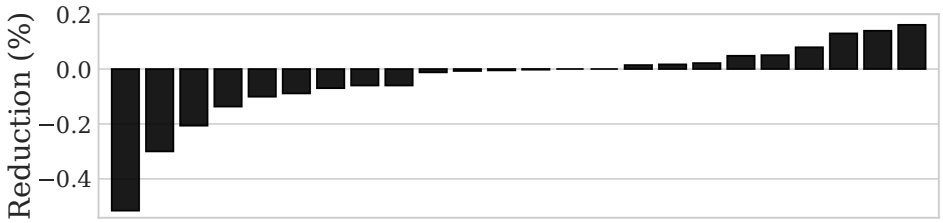


Figure 5.18: A breakdown of SalSSA[$t = 1$] on the `djpeg` benchmark. The actual contribution to the final code size for each merge operation deemed profitable by the cost model.

one of these merge operations have a very small contribution to the final code size, the profitability cost model failed enough to result in an overall code increase of about 0.3%.

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

5.3.4 Further Analysis

We also provide a breakdown showing the impact of phi-node coalescing on code size. Figure 5.19 shows the impact of our phi-node coalescing optimization technique (see Section 5.2.4). This diagram compares SalSSA to a variant without phi-node coalescing (SalSSA-NoPC) and FMSA. On average, this technique gives an additional

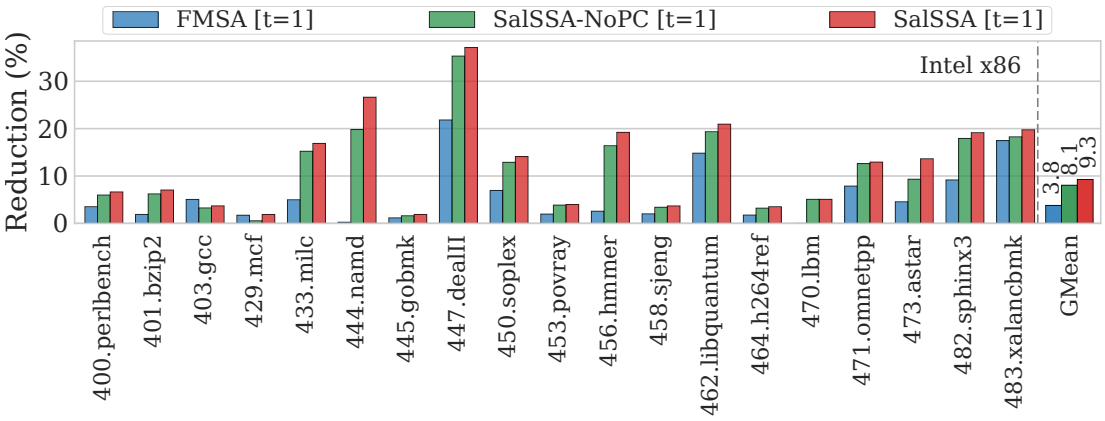


Figure 5.19: Evaluation of the impact of phi-node coalescing on the size of the final object file. SalSSA-NoPC, which includes phi-node coalescing, has a measurable benefit over the alternative without phi-node coalescing (SalSSA-NoPC). When enabled, phi-node coalescing achieves up to 7% of code size reduction on top of SalSSA-NoPC.

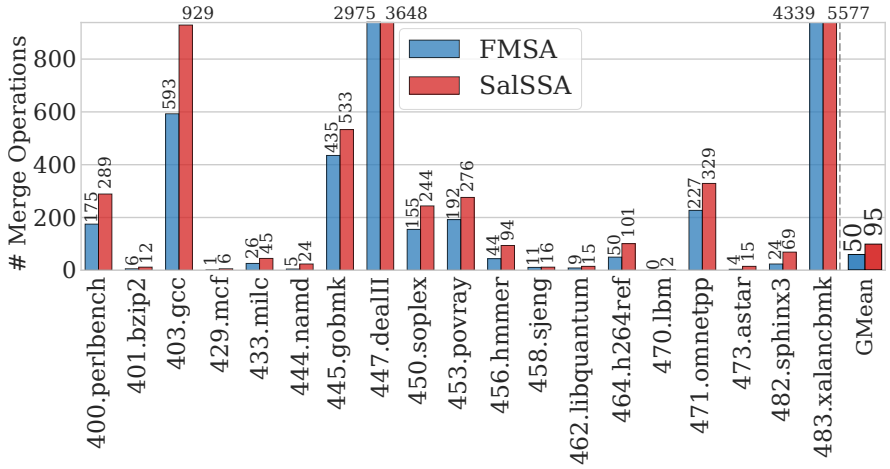


Figure 5.20: Total number of profitable merge attempts for SalSSA and FMSA on 19 SPEC CPU2006 benchmarks. For both cases, we used the lowest exploration threshold (t=1). SalSSA achieves 31% more profitable merge operations.

1.2% on code size reduction. For`444.namd`, it enables an extra 7% reduction on the code size, demonstrating the great advantage of the technique.

Figure 5.20 provides further insight into the gains of SalSSA. The figure shows the total number of profitable merging attempts for the lowest exploration threshold. While FMSA has only 9,271 profitable merge operations, SalSSA has 12,224, an increase of 31% on the number of profitable merges. Much of the improvement we observe in code size reduction comes from producing profitable merged functions where FMSA fails to gain any profit, not just from increasing the profit.

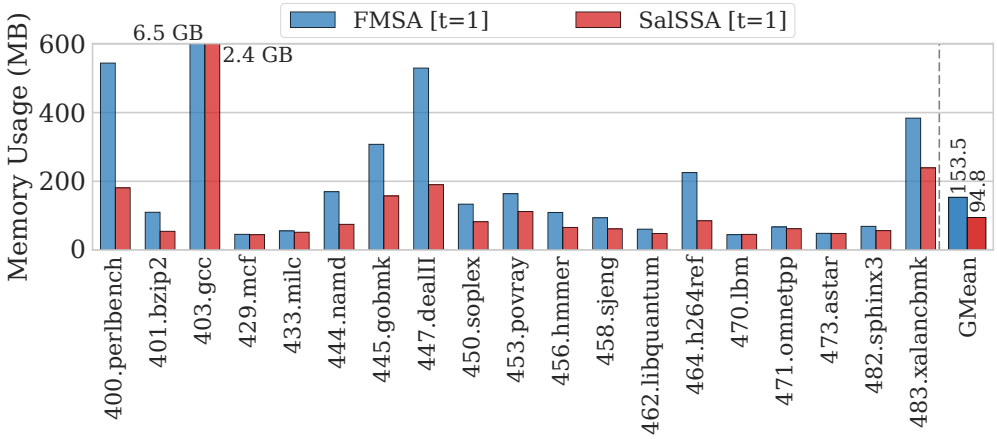


Figure 5.21: Peak memory usage during compilation time on the SPEC CPU2006 benchmark. On average, SalSSA requires less than half the memory used by FMSA.

5.3.5 Memory Usage

Because the sequence alignment algorithm [42, 46] (used by FMSA and SalSSA) has a quadratic space complexity over the length of the sequences, the difference in the size of the functions caused by register demotion translates directly to the differences in memory usage.

Figure 5.21 shows the peak memory usage across the SPEC CPU2006 suite. To isolate the impact of other compilation passes, we measure the memory usage only when running the function merging optimization. As expected, avoiding register demotion has the added benefit of lowering the memory footprint of the compilation pass. On average, SalSSA uses half the amount of memory required by FMSA. The improvements on memory usage shown in this Figure 5.21 directly reflects the difference shown in Figure 5.4.

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

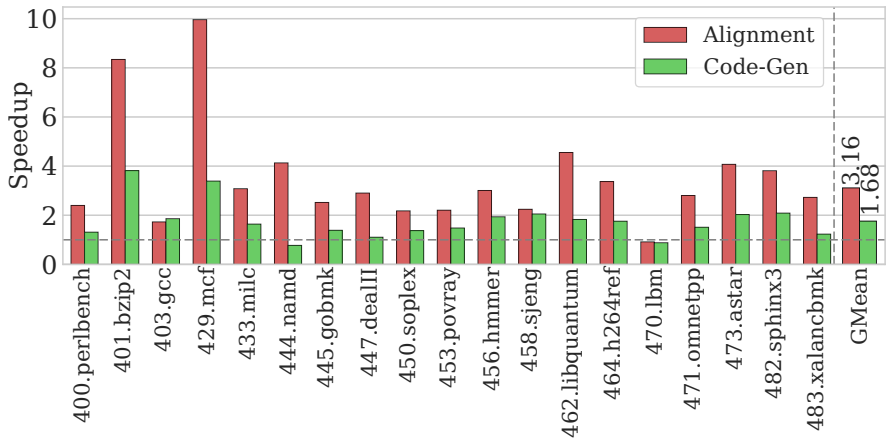


Figure 5.22: Speedup over the accumulated time spent on both sequence alignment and code generation. SalSSA produces significantly less overhead than the state-of-the-art FMSA.

5.3.6 Compilation Time Overhead

Figure 5.23 shows the normalized compile-time for *the entire* compilation process on SPEC CPU2006. The min-max bar in the diagram gives the 95% confidence interval across different compile-time measurements of a benchmark. SalSSA incurs modest compile-time overhead with an average 5% increase in the compile-time when using the lowest exploration threshold ($t = 1$). This represents a 3x reduction in the compile-time overhead compared to the 14% overhead from FMSA with the same exploration threshold. When using the largest exploration threshold ($t = 10$), we observe a 3.7x reduction in the compile-time overhead. The improvement is due to not only less time spent performing the optimization itself but also less work for the remaining compilation process since we reduce the size of the produced code. We also observe similar overhead improvement on SPEC CPU2017.

Figure 5.22 shows the speedups obtained by SalSSA for the sequence alignment and the code generator. These two stages of function merging benefit most from our techniques. As suggested earlier, both stages are accelerated because the compiler has shorter sequences to operate on under SalSSA over FMSA. The results given in Figure 5.22 and Figure 5.4 follow a very similar trend. These confirm our intuition described earlier in Section 5.1.

Since the sequence alignment algorithm is also quadratic in time over the length of the sequences, we get a quadratic speedup by avoiding register demotion with SalSSA. Code generation is linear on the size of the functions resulting in proportional speedups

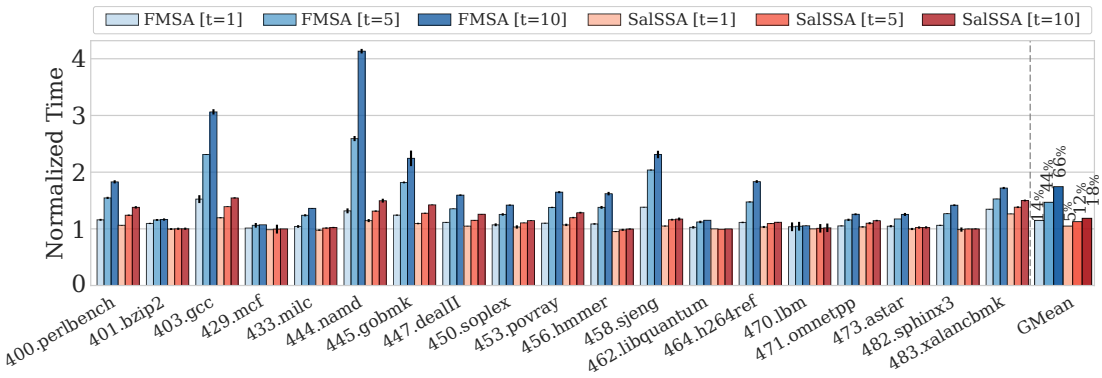


Figure 5.23: End-to-end compile-time for SalSSA and FMSA for three different exploration thresholds and 19 different SPEC CPU2006 benchmark. Compile-time is normalized to that of the baseline with no function merging. SalSSA reduces the overhead of function merging by 3× to 3.7× on average.

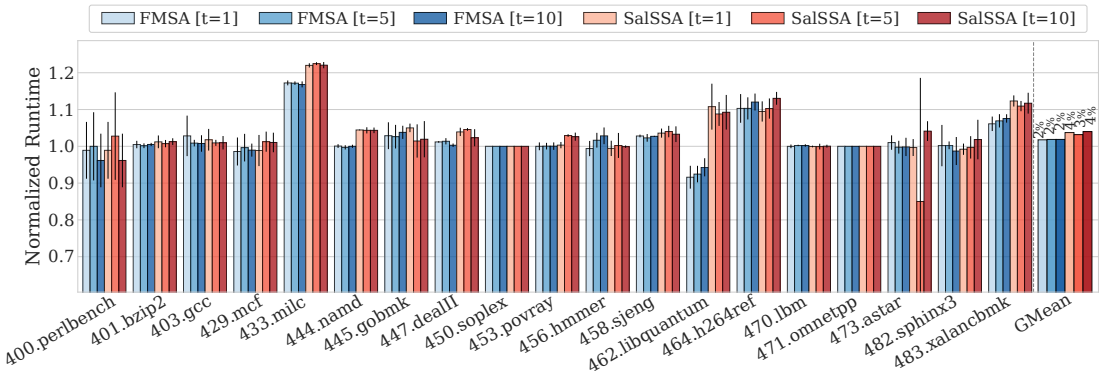


Figure 5.24: Comparison between the runtime impact from FMSA and SalSSA. Our approach increases the runtime overhead because it merges more functions. For most benchmarks, the overhead is small. For the rest, profiling-directed merging would eliminate the overhead.

in compile-time. For a couple of cases, however, the pressure put on the clean-up phase can negate those gains.

5.3.7 Performance Overhead

The primary goal of function merging is to reduce code size. Nevertheless, it is also important to keep the impact on the program runtime as low as possible. Figure 5.24 shows the normalized execution time, where the min-max bar shows the 95% confidence interval across different runs. Overall, SalSSA has an average overhead of about 4% on programs' runtime. For most benchmarks, there is no statistically significant

difference between the baseline and the optimized binary. For the rest, profiling information could be used to avoid adding overhead when mergeable code is in the most frequently executed code path.

5.4 Conclusion

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

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

Chapter 6

Avoiding Unprofitable Merge Operations with Deep Learning

Chapter **??** describes our strategy for ranking the function candidates that are more likely to be profitably merged. However, most of the merged functions are actually unprofitable candidates, since many functions are unique enough to have no profitable paring. Even if we consider only the top candidate functions, only about 13% result in a profitable merge operation.

Since the function merging operation is computationally expensive, ideally we need to focus our effort only to those are most likely to be profitable and avoid wastefully merging unprofitable pairs of functions. In this chapter, we describe our heuristic model based on deep-learning to predict whether a pair of functions can be profitably merged or not. This allows us to avoid merging pairs of functions that are unlikely to result in a profitable merge operation.

6.1 Learning a Profitability Model

6.2 Overview

Figure 6.1 provides an overview of the prediction mechanism. Our mechanism follows a similar approach to previous deep-learning techniques for tuning compilers [11, 39].

The same linearised functions used for the merge operations, as described in Chapter **??**, are used as input to the prediction model. First, we use a language model based on recurrent neural networks to encode the input functions into context vectors of fixed size. These vector encodings can be computed only once per function an cached. Fi-

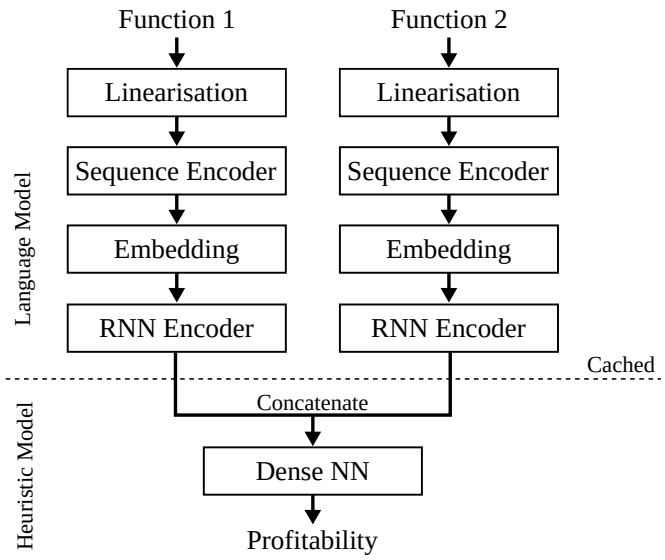


Figure 6.1: The proposed deep-learning model architecture that predicts which pairs of functions will be profitably merged. Code properties are extracted from each function into *context vectors* by the language model. These context vectors are cached to be later fed to the heuristic model to produce the final profitability prediction.

nally, the context vectors of two input functions are concatenated and fed to a dense deep neural network to classify whether or not those functions are profitably merged.

Chapter 7

Conclusion

This thesis presents a novel compiler optimisation for reducing code size by merging functions. Chapter **??** describes our novel approach, based on sequence alignment, for merging any two functions. Chapter **??** describes how our function merging technique can be combined with a optimisation strategy in order to search for profitable functions to merge, which includes a profitability cost model and a ranking of candidate functions.

This chapter is structured as follows: Section 7.1 summarises the main contributions of this thesis, Section 7.2 describes future research directions, and finally Section 7.3 provides concluding remarks.

7.1 Contribution

This thesis presents a novel function merging technique alongside an optimisation strategy. Our novel technique uses sequence alignment algorithms from bioinformatics in order to identify the similarity between functions being merged. The proposed optimisation is very effective at reducing code size by merging arbitrary functions. Our approach does not suffer from any of the major limitations of existing solutions, outperforming them by more than 2.4×. We also proposed a ranking-based exploration mechanism to focus the optimization on promising pairs of functions. Ranking reduces the compilation-time overhead by orders of magnitude compared to an exhaustive quadratic exploration. With this framework, our optimization is able to reduce code size by up to 25%, with an overall average of about 6%, while introducing an average compilation-time overhead of only 15%. Coupled with profiling information, our optimization introduces no statistically significant impact on performance.

7.2 Future Work

For future work, we plan to focus on improving the ranking mechanism to reduce compilation time. In order to avoid code size degradation, we also plan to improve the compiler's built-in static cost model for code size estimation. We also plan to work on the linearisation of the candidate functions, allowing instruction reordering to maximize the number of matches between the functions. Finally, we also plan to incorporate instruction reordering into function merging to maximize the number of matches between the functions regardless of the original code layout. One can also investigate the application of phi-node coalescing outside function merging. We envisage further improvements can be achieved by integrating the function-merging optimization to a summary-based link-time optimization framework, such as ThinLTO in LLVM. As a future work, we can also analyze the interaction between function merging and other optimizations such as inlining, outlining, and code splitting.

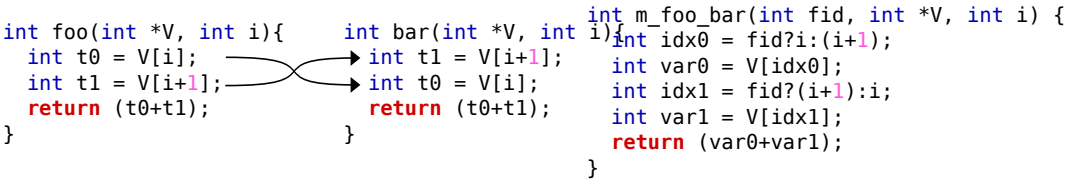
7.2.1 Better Code Generator

In order to better compress code, one can improve the code generator, optimising the parameters used as function identifiers based on calling conventions, exploit target specific instructions, and optimize operand selection and branch instructions that result from merging code. There are also many missing features in the code generator that are important for its application in the industry. For example, a new code generator could also be able to appropriately handle variable length arguments, and debug information. When merging functions, having an optimised code generator is as important as optimally identifying which instructions to merge.

7.2.2 Handling Code Reordering

All existing function merging techniques rely on the order in which instructions appear inside the basic blocks and their arrangement, failing to profitably merge equivalent functions when confronted even with the smallest variations on the ordering of instructions and basic blocks. Figure 7.1a shows an example of two functions that all existing techniques fail to merge even though they are obviously identical. Our current technique produces the merged function shown in Figure 7.1b, which is considered unprofitable as it is unable to reduce code size. Note that all indexing computation is duplicated, including the increment operation, resulting in a merged function that

is unnecessarily bigger than the two original functions together. We need more powerful graph, rather than sequence, alignment techniques to better identify and merge differently ordered but semantically equivalent code.



(a) Two equivalent functions.

(b) Merged function currently produced by FMSA.

Figure 7.1: Example of how even trivial reordering is poorly handled by the existing solutions.

7.2.3 Merging Across All Scopes

Existing techniques are limited to one particular scope. While function merging is applied only to whole functions, function outlining is commonly applied at the basic block level. However, equivalent code can be found within or across functions, which themselves may reside in the same source file or be spread across multiple source files. Therefore, we need to develop a novel unified optimization capable of merging semantically equivalent code that can span anything between a single basic block up to a whole function. This unification has the extra benefit of also addressing the phase ordering problem by coordinating the merge operations on different scopes.

7.2.4 Scaling for Large Programs

Although our optimisation achieves very good results in terms of code compression, it is still unable to handle large programs in a real scenario. Its time complexity and memory usage requirements would prevent it from optimizing large programs such as web browsers, Clang/LLVM, and operating systems, as these programs tend to have many functions with several thousands of instructions. Link-time optimization (LTO) makes this matter even worse by optimizing the code after the whole program has been linked into a single module, imposing a huge pressure on memory usage and compilation time.

The optimization of different translation units can be distributed across different machines and merge operations locally performed in parallel. In order for this to work,

an important challenge that needs to be addressed concerns ranking and merging functions that reside in different translation units. However, this is essential to enable the use of LTO on real programs while keeping the memory usage and compilation time acceptable.

7.2.5 Powered by Deep Learning

In order to reduce compilation time while also being effective, the ranking-based exploration mechanism tries to efficiently focus the search only to the most promising pairs of functions. However, the existing solution is still very wasteful as most of the merged functions are discarded by the profitability analysis. Identifying what would be profitably merged is a very challenging task.

Given our group's expertise on the area, we plan to use solutions based on deep learning to efficiently predict which pairs of functions are most likely to be profitable to merge. This approach has the potential to reduce compilation time even further while also improving the compression by finding profitable candidates that are currently missed. Having accurate target-specific cost models is crucial for the effectiveness of the profitability analysis. We plan to explore the use of machine learning techniques to develop more accurate cost models.

One can investigate the use of deep learning to align two functions and better identifying what can be merged. Sequence alignment focusses only on maximizing the number of merged instructions, without necessarily minimizing the number of operand selections or branches. A smarter approach that understands how instructions interact with each other would be very beneficial.

7.2.6 Avoiding Performance Overheads

For many real applications, it is desirable to achieve a good balance between code size and performance. Preliminary results show that performance degradation can be completely avoided by using profiling to guide the merging decisions. One can avoid adding branches inside hot execution paths, therefore avoiding performance penalties. Although hot code can be merged, it is important to minimize unnecessary branches when merging hot code. We will develop a profile-guided optimization that automatically identify the best trade-off between code-size and performance.

7.2.7 Less Memory Usage by JIT

Ahead-of-time and just-in-time (JIT) compilation have completely different requirements. Function merging can be used to reduce the amount of memory used by programs running on a JIT environment. However, our solution must be adapted to the requirements that are specific to a JIT environment, as JIT compilers have the extra challenge of having to optimize the code as fast as possible. This would require the development of completely new algorithms for ranking and aligning functions that are suitable for this application domain. Another possibility is to exploit the fact multiple programs may be simultaneously running on the same JIT environment and merge code across different programs, reducing the overall memory usage.

7.3 Summary

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] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232, 2017.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

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

[38] 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.

[39] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

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

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

[42] 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.

[43] 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.

[44] 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.

[45] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. Look-ahead SLP: Auto-vectorization in the presence of commutative operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 163–174, New York, NY, USA, 2018. ACM.

[46] 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.

[47] 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.

[48] 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.

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

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

[51] 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.

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

[53] 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.

[54] 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.

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

[56] 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.