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. 2019.

(*Rodrigo Caeteano de Oliveira Rocha*)

Table of Contents

1 Introduction **1**

1.1 Sources of Similarities 1

2 Related Work **3**

2.1 Identical Code Folding in Linkers 3

2.2 Identical Function Merging 3

3 Function Merging **5**

3.0.1 Linearization 6

3.0.2 Sequence Alignment 7

3.0.3 Equivalence Relation 8

Bibliography **11**

Chapter 1

Introduction

1.1 Sources of Similarities

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

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

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

After template specialization and applying optimizations:

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

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

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

Chapter 2

Related Work

2.1 Identical Code Folding in Linkers

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

2.2 Identical Function Merging

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

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

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

Chapter 3

Function Merging

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

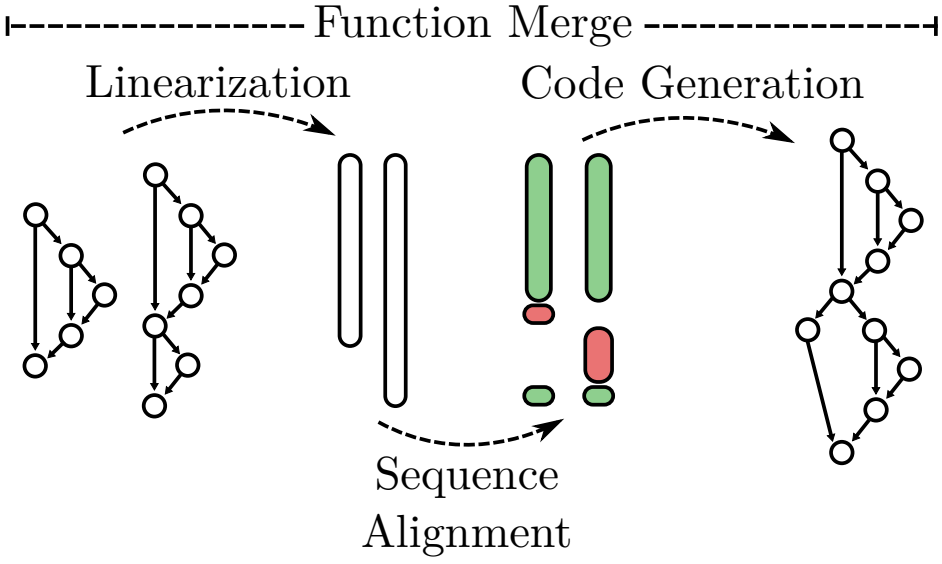

Figure 3.1: Overview of our function-merging technique.

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

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

3.0.1 Linearization

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

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

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

3.0.2 Sequence Alignment

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

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

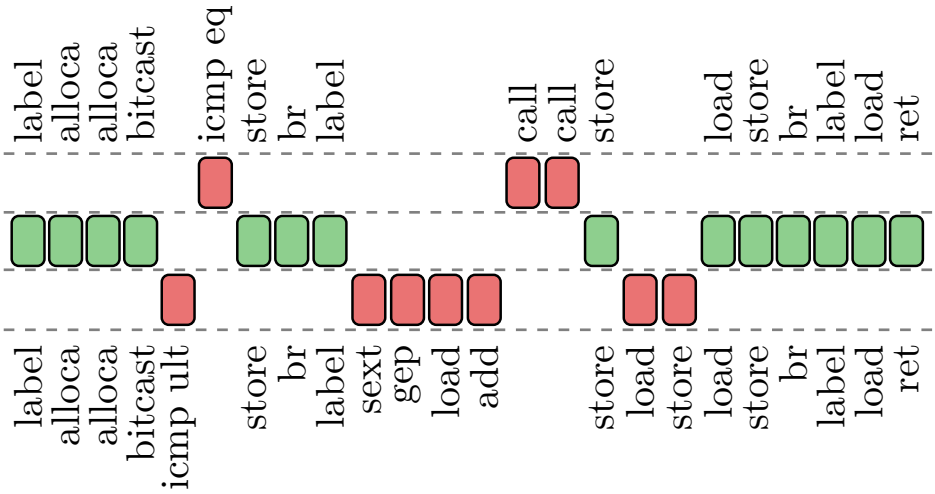

Figure 3.2: The sequence alignment between two functions.

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

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

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

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

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

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

3.0.3 Equivalence Relation

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

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

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

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

3.0.3.1 Handling Exception Handling Code

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

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

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

Bibliography

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

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

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

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

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

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