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

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.

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.