HyFM: Function Merging for Free

Rodrigo C. O. Rocha
University of Edinburgh
United Kingdom
rrocha@ed.ac.uk

Pavlos Petoumenos
University of Manchester
United Kingdom
pavlos.petoumenos@manchester.ac.uk

Zheng Wang
University of Leeds
United Kingdom
z.wang5@leeds.ac.uk

Murray Cole
University of Edinburgh
United Kingdom
mic@inf.ed.ac.uk

Kim Hazelwood
Facebook AI Research
United States
kimhazelwood@fb.com

Hugh Leather
Facebook AI Research
United States
hleather@fb.com

Abstract

Function merging is an important optimization for reducing code size. The existing state-of-the-art relies on a well-known sequence alignment algorithm to identify duplicate code across whole functions. However, this algorithm is quadratic in time and space on the number of instructions. This leads to very high time overheads and prohibitive levels of memory usage even for medium-sized benchmarks. For larger programs, it becomes impractical.

This is made worse by an overly eager merging approach. All selected pairs of functions will be merged. Only then will this approach estimate the potential benefit from merging and decide whether to replace the original functions with the merged one. Given that most pairs are unprofitable, a significant amount of time is wasted producing merged functions that are simply thrown away.

In this paper, we propose HyFM, a novel function merging technique that delivers similar levels of code size reduction for significantly lower time overhead and memory usage. Our alignment strategy works at the block level. Since basic blocks are usually much shorter than functions, even a quadratic alignment is acceptable. However, we also propose a linear algorithm for aligning blocks at a much lower cost. We extend this strategy with a multi-tier profitability analysis that bails out early from unprofitable merging attempts. By aligning individual pairs of blocks, we are able to decide their alignment's profitability before actually generating code.

Experimental results on SPEC 2006 and 2017 show that HyFM needs orders of magnitude less memory, using up to 48 MB or 5.6 MB, depending on the variant used, while the state-of-the-art requires 32 GB in the worst case. HyFM also runs over 4.5× faster, while still achieving comparable code size reduction. Combined with the speedup of later compilation stages due to the reduced number of functions, HyFM contributes to a reduced end-to-end compilation time.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Keywords: Code-Size Reduction, Function Merging, LLVM, Link-Time Optimization, Interprocedural Optimization

ACM Reference Format:
Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function Merging for Free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3461648.3463852

1 Introduction

While often overlooked, program size can be a first-order constraint. Regardless of the type of the system, from IoT devices up to cloud servers, they are all operating under limited addressable memory, storage, or bandwidth. When the program becomes excessively large relative to the given constraints, this has a detrimental effect on the system. In the extreme, this means failure. This is very likely to happen as programs gain new features over time, continuously growing in size and complexity [4, 14]. In such scenarios, reducing the application footprint is essential [3, 4, 11, 25, 26, 30].

One important class of optimizations for reducing code size is function merging. Existing techniques range from simple passes merging identical functions at the compiler intermediate representation (IR) [2, 15] or the binary level [1, 13, 28] up to approaches that identify and merge similar subsequences in otherwise dissimilar functions [9, 23, 24]. As already noted by Chabbi et al. [4], these techniques have either limited benefit on code reduction or unacceptable compilation overheads for production, especially considering builds using link-time optimizations (LTO) where inter-procedural

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES '21, June 22, 2021, Virtual, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8472-8/21/06…$15.00
https://doi.org/10.1145/3461648.3463852

optimizations have greater opportunities but at a greater cost [10].

The current state-of-the-art, SalSSA [23, 24], achieves on average a 10% code size reduction but at the cost of crippling compile-time inefficiencies. In this paper, we show that SalSSA can lead to 40% slower compilation, taking up to 32 GB of memory for temporary data when compiling a modestly-sized program. Such a resource requirement is beyond what is typically available to a developer and thus unsuitable for optimizing real-life programs.

These inefficiencies stem directly from SalSSA's core innovation, i.e., the sequence alignment algorithm used to identify mergeable instructions in a pair of input functions. The alignment algorithm has quadratic time and space complexity, so applying it on whole functions with thousands of instructions results in unacceptable overheads. This severely limits the applicability of function merging on relatively large programs. To make function merging scalable and practical, we need to find ways to significantly reduce the memory and compilation overhead. Our work is designed to offer such capabilities.

In this paper, we present HyFM, a novel function merging technique that addresses the performance inefficiencies of SalSSA. Our main insight is that most of the code reduction of SalSSA comes from matching highly similar basic blocks. Even though it is able to align arbitrary subsequences spanning basic block boundaries, profitable alignments usually contain instructions from one block matched to instructions from a single other block. We show that an approach which quickly identifies similar basic blocks and then aligns their short instruction sequences achieves a similar code reduction for a much lower overhead.

More specifically, our solution is three fold:

- We align the input functions on a per basic block manner. First, we pair similar basic blocks by minimizing the distance between their fingerprints. Then, we only align the instructions within each pair of basic block. Even with a quadratic alignment algorithm, basic blocks are usually much shorter than functions, translating into a much faster alignment.
- We propose a linear pairwise alignment as an alternative to the quadratic one. For highly similar basic blocks, it achieves similar results but has negligible time and space overheads.
- We estimate the profitability of the aligned basic blocks before actually generating their merged code. If unprofitable, we ignore them, improving the overall profitability of the whole merged function and simplifying code generation. If all paired blocks in a pair of functions are unprofitable, we skip merging the function pair altogether, speeding up the optimization process compared to SalSSA.

Experimental results on SPEC CPU 2006 and 2017 show that HyFM runs over 4.5× faster than SalSSA. Compared to a baseline without function merging, HyFM reduces end-to-end compilation time by up to 18% and 2.1% on average. HyFM also has orders of magnitude lower peak memory usage, using up to 48 MB or 5.6 MB, depending on the variant used, while SalSSA requires 32 GB in the worst case. We achieve all these compilation-time benefits without degrading its ability to reduce code size.

2 Background and Motivation

In this section, we first introduce the working mechanism of SalSSA [24], the state-of-the-art function merging technique. We highlight the main drawbacks of SalSSA in terms of compile time and memory footprint. We then outline how we can address these drawbacks without compromising on code size reduction.

2.1 Function Merging via Sequence Alignment

Existing function merging techniques consist of three major stages: choosing which functions to merge, producing the merged function, and estimating the merging profitability.

In order to pair similar functions for merging, SalSSA employs a ranking strategy based on the similarity of the *fingerprints* of the functions. A fingerprint summarizes the content of a function as a fixed-size vector of the frequency of each LLVM-IR opcode. The representation allows the compiler to compare functions using a simple distance metric, such as the Manhattan distance. For a given reference function, all other functions are ranked based on their distance and the closest function is chosen for merging.

Merging two functions requires identifying similar code segments in the two functions that can be profitably merged. The main innovation of SalSSA [24] and its predecessor [23] is the use of a sequence alignment algorithm, called the Needleman-Wunsch algorithm, for identifying similar code segments. This allows them to merge arbitrary pairs of functions. First, they transform each function into a linear sequence of labels and instructions. Then, the alignment algorithm is applied on the sequences of the whole input functions. The resulting alignment is used to generate the merged function. Once the merged function has been generated, they apply an SSA reconstruction algorithm. For a final clean up, they simplify the merged function by removing redundant instructions introduced by function merging.

Finally, a profitability analysis estimates the benefit of replacing the original pair of functions with the simplified merged function. If unprofitable, the merged function is simply thrown away. Otherwise, they delete the original functions, redirecting the calls to the merged function.

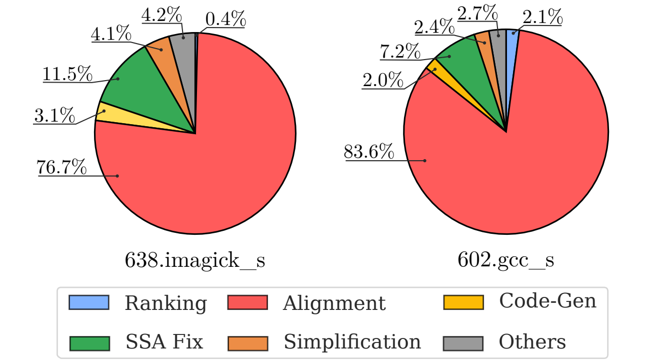


Figure 1. Breakdown of the relative runtime for the different stages from SalSSA. Alignment takes 25 seconds and 4.2 minutes on `638.imagick_s` and `602.gcc_s`, respectively.

2.2 Limitations of the State of the Art

After further investigation, we observed that SalSSA was unable to optimize `602.gcc_s`, from SPEC 2017, due to an out of memory crash. Our machine with 16 GB of memory was not enough to handle SalSSA. We succeeded only after migrating to a 64 GB machine which could fit the 32 GB of temporary data produced by function merging. We realized that this is due to the quadratic algorithm used for aligning the two functions selected for merging. Because this algorithm is applied on the linearized sequences of the whole input functions, SalSSA incurs a high memory footprint when merging even medium sized functions. For larger ones, it is impossible to apply it on most workstations or even many servers, making SalSSA impractical for use in production.

For the same reason, alignment brings the compilation process to a crawl for large functions. Figure 1 shows the running time breakdown for the different stages of the function merging pass in the LLVM-based SalSSA implementation for two SPEC CPU2017 benchmarks. Sequence alignment dominates the running time of function merging, representing up to 83% of its overall running time. Sequence alignment alone takes 25 seconds and 4.2 minutes on `638.imagick_s` and `602.gcc_s`, respectively. The alignment stage also causes the peak in memory usage for these two programs, 4.5 GB for `638.imagick_s` and 32 GB for `602.gcc_s`. This is not surprising, as the Needleman-Wunsch algorithm has a quadratic complexity in both time and memory usage. Because this algorithm is applied on linearized sequences of the whole input functions, programs containing large functions, such as the ones in our example, are heavily affected.

Most of the rest of the running time of function merging is associated with producing merged functions from the aligned sequences. This includes the time spent on the code generation stage (Code-Gen), SSA reconstruction (SSA Fix), and code simplification (Simplification). These stages account for 18.7% of the SalSSA's running time on `638.imagick_s` and 11.6% on `602.gcc_s`. However, for other programs, these stages may represent the vast majority of SalSSA's running time (see Section 4.3).

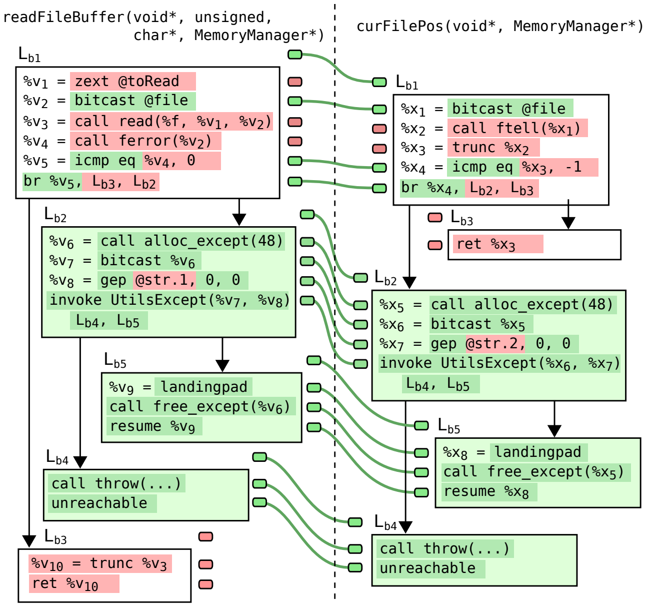
This breakdown includes the cost for producing both *profitable* and *unprofitable* merged functions. In fact, most of it is wasted on merged functions that will be rejected by the profitability analysis. These costs are pronounced because unprofitably merged functions have no limit on their size or complexity, often adding a significant pressure on the SSA reconstruction and simplification stages. This effect is tied to the alignment strategy, since a good alignment is needed for producing profitably merged functions. As we discuss in Section 2.3, a better approach would include a finer grain profitability analysis that would allows us to bail out from merging complex and unprofitable code as early as possible.

2.3 When Less Is More

We observe that most of the benefit of function merging often comes from merging highly similar, but not necessarily identical, basic blocks. Figure 2 shows one such example extracted from the `483.xalancbmk` benchmark found in SPEC CPU2006. This example shows the two input functions annotated with the alignment produced by SalSSA. Merging these two functions contributes to a reduction of 33 bytes in the final object file.

While this approach is flexible enough to identify very complex alignments, what it actually produces is three aligned pairs of basic blocks and a few aligned instructions in the entry blocks. More importantly, these entry block instructions offer nothing in terms of code size reduction. The gains of merging them are negated by the extra branches and operand selections needed to preserve the program's semantics. Since SalSSA analyzes the profitability of the final merged function as a whole, this unprofitable sequence of instructions will be merged because of the three highly profitable basic blocks. For the same reason, we may have profitable areas of code rejected because the rest of the merged function is unprofitable.

This example shows us that we could achieve similar code size reduction by breaking the problem of aligning functions into two simpler processes: first identifying highly similar basic blocks and then aligning the instructions in each pair of similar blocks. By operating on basic blocks, we could greatly reduce the length of the sequences to be aligned and the associated compilation and memory overhead. Furthermore, by making profitability decisions for each pair of basic blocks separately, we could avoid merging unprofitable pairs. The rest of this paper shows how we use such an approach to overcome the weaknesses of SalSSA and make function merging practical for optimizing large programs.



```
readFileBuffer(void*, unsigned,
                  char*, MemoryManager*)
Lb1
%v1 = zext @toRead
%v2 = bitcast @file
%v3 = call read(%f, %v1, %v2)
%v4 = call ferror(%v2)
%v5 = icmp eq %v4, 0
br %v5, Lb3, Lb2
Lb2
%v6 = call alloc_except(48)
%v7 = bitcast %v6
%v8 = gep @str.1, 0, 0
invoke UtilsExcept(%v7, %v8)
    Lb4, Lb5
Lb5
%v9 = landingpad
call free_except(%v6)
resume %v9
Lb4
call throw(...)
unreachable
Lb3
%v10 = trunc %v3
ret %v10
```

```
curFilePos(void*, MemoryManager*)
Lb1
%x1 = bitcast @file
%x2 = call ftell(%x1)
%x3 = trunc %x2
%x4 = icmp eq %x3, -1
br %x4, Lb2, Lb3
Lb3
ret %x3
Lb2
%x5 = call alloc_except(48)
%x6 = bitcast %x5
%x7 = gep @str.2, 0, 0
invoke UtilsExcept(%x6, %x7)
    Lb4, Lb5
Lb5
%x8 = landingpad
call free_except(%x5)
resume %x8
Lb4
call throw(...)
unreachable
```

Figure 2. Example extracted from `483.xalancbmk` in SPEC CPU2006. Instructions marked green have been aligned through sequence alignment with an instruction from the other function. SalSSA would attempt merging all matched instructions but only the ones in fully aligned basic blocks would be profitable.

3 Hybrid Function Merging

In this section, we propose HyFM (Hybrid Function Merging), a novel function merging technique that can operate on all functions regardless of their size with little to no compilation overheads. To achieve this goal, we rely on the insights discussed in Section 2. Our solution is three-fold: 1) We introduce an alignment strategy that works on the level of basic blocks, without crossing their boundaries, leading to faster and less memory demanding alignment; 2) We incorporate a multi-tier profitability analysis that allows us to bail out from unprofitable merging attempts even before code generation; 3) We introduce a linear pairwise alignment for basic blocks of the same size that produces good results on highly similar blocks. This technique can be enabled as an alternative to the quadratic sequence alignment algorithm. Both techniques have their place, offering different trade-offs.

3.1 Overview

For all candidate functions and basic blocks, we generate a fixed-vector representation, namely, their fingerprint [23, 24]. We match each function with its most similar available function, the one with the shortest fingerprint distance. Instead of aligning their linearized representations directly, we work at the basic block level. We pair similar basic blocks of the two functions based on their fingerprint distances. We align the instructions in these paired basic blocks using either the Needleman-Wunsch alignment [18] or our linear pairwise alignment strategy. We employ the first-tier profitability analysis on each alignment. If the cost model deems it unprofitable, we skip the pair. The pairing of basic blocks, the alignment, and the first-tier profitability analysis are executed in rounds, in a greedy manner. That is, the first profitable pairing is taken, however, unprofitable paired blocks are freed for another pairing, if necessary.

Once all basic blocks have been processed, we combine the block alignments into a function-wide one and we produce the merged function using the same code generation proposed for SalSSA [24]. If no profitable pair of basic blocks was found, we bail out before code generation. Finally, we perform the second-tier profitability analysis, which is the same used by SalSSA, to decide whether replacing the original functions by the merged one reduces code size. If not, we reject the merged function and we keep the original ones.

For brevity, the rest of the discussion will focus on how HyFM differs from previous approaches.

3.2 Pairing Similar Basic Blocks

We pair similar basic blocks based on distance of their fingerprints. This pairing process is similar to the search strategy used for pairing functions [23]. We use the same fingerprint, a fixed-size vector of integers with the frequency count of each opcode. It can be used to represent any piece of code, from basic blocks to whole functions.

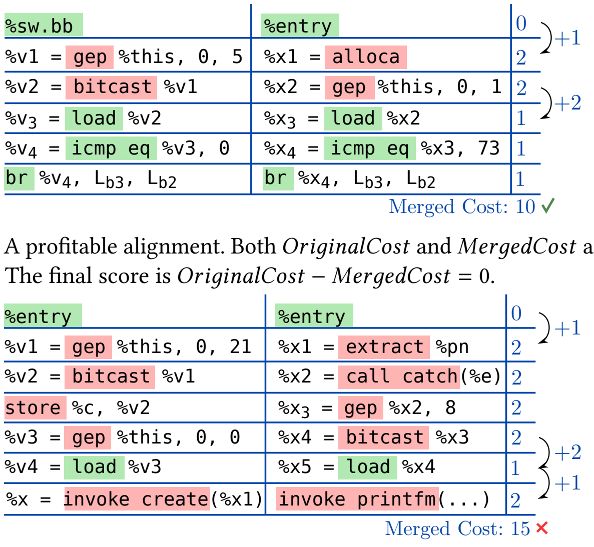
The overall idea is that for each block in one function we select a block from the other function that minimizes the Manhattan distance between their fingerprints. Formally, given a block $B_1 \in F_1$, where F_1 is the set of all blocks from function one, B_1 is paired with a block $B_m \in F_2$ such that:

$$d(B_1, B_m) = min\{d(B_1, B_2) : B_2 \in F_2\}$$

where $d(B_1, B_2)$ represents the distance between the fingerprints of the basic blocks B_1 and B_2.

After pairing two basic blocks, B_1 and B_2, they have their instructions aligned (see Section 3.3) and their merging profitability estimated (see Section 3.4). If they are deemed profitable, both blocks are removed from their respective working list. Otherwise, only B_1 is removed from the working list of blocks from F_1, i.e., B_2 can still be paired with another block, but not B_1. In other words, basic blocks from function F_1 are paired only once, even if its alignment is deemed unprofitable. As a result, given two input functions, this pairing process is quadratic on their number of basic blocks. This number is usually much smaller than the number of instructions in the function, so the cost of pairing is much lower than the cost of aligning whole functions in SalSSA, despite both being quadratic. For very large numbers of basic blocks, efficient nearest neighbor search techniques could keep the cost low but this was not needed in our experiments.

HyFM pre-computes the fingerprint of every basic block in the input functions, which is a single linear cost over all their basic blocks and instructions. Meanwhile, the distance



%sw.bb	%entry	0	+1
%v1 = gep %this, 0, 5	%x1 = alloca	2	
%v2 = bitcast %v1	%x2 = gep %this, 0, 1	2	+2
%v3 = load %v2	%x3 = load %x2	1	
%v4 = icmp eq %v3, 0	%x4 = icmp eq %x3, 73	1	
br %v4, Lb3, Lb2	br %x4, Lb3, Lb2	1	

Merged Cost: 10 ✓

(a) A profitable alignment. Both *OriginalCost* and *MergedCost* are 10. The final score is *OriginalCost* − *MergedCost* = 0.

%entry	%entry	0	+1
%v1 = gep %this, 0, 21	%x1 = extract %pn	2	
%v2 = bitcast %v1	%x2 = call catch(%e)	2	
store %c, %v2	%x3 = gep %x2, 8	2	
%v3 = gep %this, 0, 0	%x4 = bitcast %x3	2	+2
%v4 = load %v3	%x5 = load %x4	1	+1
%x = invoke create(%x1)	invoke printfm(...)	2	

Merged Cost: 15 ✗

(b) An unprofitable alignment. *OriginalCost* is 0 and *MergedCost* is 15. The final score is −3. A negative score means it is unprofitable.

Figure 3. Two examples of the pairwise alignment. Only instructions in corresponding positions are aligned. Instructions match if they have the same opcode.

between two fingerprints is computed in constant time, since the number of opcodes is a small constant.

3.3 Aligning Paired Basic Blocks

Basic blocks already represent a linearized sequence of instructions. Any sequence alignment algorithm can be used on them the same way they can be used on linearized functions. The Needleman-Wunsch [18] algorithm used by SalSSA remains a good choice. It may be quadratic in both space and time on the length of the sequences but basic block sequences are usually much shorter than functions, making the cost of alignment lower than in previous approaches.

Our observations in Section 2, though, indicate that a quadratic algorithm might be an excessive solution. Profitable sequences tend to be highly similar, so aligning them is usually straightforward and a simpler approach should suffice. Based on this insight, we implemented a linear alignment algorithm. Its assumption is that profitable pairs of blocks are almost identical in terms of opcodes differing only in a few individual cases. This translates into a pairwise alignment of same size basic blocks where only corresponding instructions in the two blocks can match. Figure 3 illustrates two examples of basic blocks aligned using our strategy. It also includes the costs estimated by our profitability analysis, which we discuss in Section 3.4.

Restricting alignment to basic blocks of the same size has the added benefit that it simplifies the pairing strategy. We only have to consider fingerprints for basic blocks of the same size, so we group them by block size and we restrict our search in the right group. In the worst case, all basic blocks would have the same size and the search would remain quadratic on the number of basic blocks, as discussed in Section 3.2. However, this is unlikely to happen in large functions, which is where the number of basic blocks might be a problem. Overall, this solution is lean on memory usage and usually the fastest for aligning paired basic blocks, as corroborated by our evaluation in Section 4.

3.4 Multi-tier Profitability Analysis

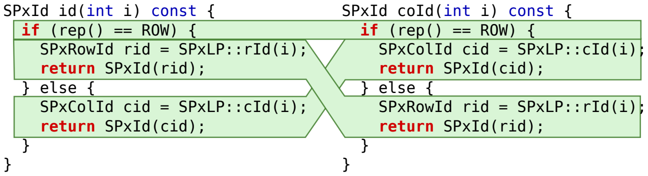
HyFM incorporates a multi-tier profitability analysis that enables it to bail out early from an unprofitable merge operation. The first tier consists of a simple analysis applied on each pair of basic blocks selected for alignment, either accepting or rejecting the alignment between two blocks. The second tier consists of the same profitability analysis that is also performed by prior techniques, i.e., FMSA and SalSSA, which is responsible for evaluating whether the merged function is smaller than the original input functions.

The last column of Figure 3 shows how the first tier analysis is employed alongside the pairwise alignment strategy. The same analysis can also be applied on pairs of basic blocks aligned using the Needleman-Wunsch algorithm. The analysis tries to estimate the cost of merging, the total number of instructions that will be necessary for merging the aligned blocks. If two instructions match, then a single instruction is needed (i.e., a cost of +1 is assigned to this entry). If they mismatch, then both instructions are needed (i.e., a cost of +2 is assigned to this entry). Moreover, we need extra instructions to transition from matching subsequences to mismatching ones, and vice versa. This is represented by the arrows in Figure 3. One branch instruction is needed to split control flow into two mismatching instructions, while two branch instructions are needed to join it back into a matching pair of instructions. The *MergedCost* is the sum of all these costs. The profitability score is defined as *OriginalCost* − *MergedCost*, where *OriginalCost* is simply the number of instructions in the original basic blocks. Therefore, a negative profitability score means that merging those two basic blocks is unprofitable. When this is the case, we ignore the alignment.

By rejecting individual basic block alignments, we are able to decide early whether merging a pair of functions might be profitable. If we rejected all block alignments, then by definition there is no point in merging the functions. Previous approaches, without a first tier analysis, have to rely on the second tier exclusively which is applied after the functions are merged.

3.5 Independence from Code Layout

Unlike all prior techniques, HyFM is able to merge similar basic blocks regardless of their position in the control-flow graphs from the input functions. Figure 4 shows an example of two functions that all prior techniques fail to merge even though they are highly similar.



```
SPxId id(int i) const {
  if (rep() == ROW) {
    SPxRowId rid = SPxLP::rId(i);
    return SPxId(rid);
  } else {
    SPxColId cid = SPxLP::cId(i);
    return SPxId(cid);
  }
}
```

```
SPxId coId(int i) const {
  if (rep() == ROW) {
    SPxColId cid = SPxLP::cId(i);
    return SPxId(cid);
  } else {
    SPxRowId rid = SPxLP::rId(i);
    return SPxId(rid);
  }
}
```

Figure 4. Example with code reordering extracted from the `450.soplex` program.

Due to their rigid linearization strategy, both FMSA and SalSSA are unable to properly match the basic blocks of the if-else structure, resulting in sub-optimal merged functions that are deemed unprofitable. Their linearization traverses the control-flow graph in a canonical manner, preventing blocks from being rearranged for a better merging [23, 24]. Meanwhile, earlier techniques fail to merge this example as they are restricted to functions with identical control-flow graphs where corresponding blocks are merged [2, 9, 15].

HyFM is able to correctly pair these basic blocks. Because the basic blocks are rearranged, the label operands of the conditional branch need to be handled in order to preserve the program semantics. For swapped label operands, HyFM simply uses the optimized operand resolution proposed by SalSSA, where an `xor` operation is applied on the condition of the branch and the function identifier [24].

4 Evaluation

In this section, we compare HyFM against the state-of-the-art function merging technique, SalSSA [24]. First, we evaluate the code size reduction achieved by each technique, demonstrating that our approach is on a par with SalSSA. Then we show that HyFM reduces significantly the overhead of function merging. Combined with the speedup in later stages of the compilation pipeline due to the reduced amount of code, HyFM leads to faster end-to-end compilation than a baseline with no function merging enabled. Finally, we demonstrate how our contributions reduce the memory usage by several orders of magnitude.

4.1 Experimental Setup

In addition to evaluating SalSSA, we also consider four variations of our technique based on two dimensions: 1) the linear Pairwise Alignment (PA) versus the quadratic Needleman-Wunsch Alignment (NW), both on a per basic block manner and 2) using a multi-tier profitability analysis versus using only the standard profitability analysis from SalSSA, which is the analysis applied on the whole function after generating the merged function. As described in Section 3, the [PA] variant is, by construction, limited to merging only basic blocks of the same size. The [NW] variant can merge blocks of different sizes. The four variations are:

- [PA]: PA with the Multi-tier Profitability analysis.
- [PA,NMP]: PA with No Multi-tier Profitability.
- [NW]: NW with the Multi-tier Profitability.
- [NW,NMP]: NW with No Multi-tier Profitability.

For SalSSA, we used the version published in the evaluation artifact [22]. To keep the comparison fair, we implemented HyFM for the same compiler version as SalSSA, namely, LLVM v11. We evaluated all techniques on the C/C++ programs from the SPEC CPU 2006 and the SPEC CPU 2017 benchmark suites [27][1]. The baseline in all cases is the LLVM build in full LTO mode without any function merging.

We target the Intel x86 architecture. All experiments were performed 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, we repeated all compilation and runtime overhead experiments 5 times. We report the average values and their 95% confidence intervals.

We evaluate all approaches in terms of code size reduction, time overhead of function merging, end-to-end compilation time, and peak memory usage. To better examine the tradeoff between code size reduction and compilation time, we also introduce and measure a new metric called *average reduction speed* which shows the efficiency of the optimization at reducing code size. This metric offers a single number that allows us to compare how different versions address the trade-off between compilation time and code size reduction.

Definition 4.1 (Average Reduction Speed)**.** For a given input program and optimization, let S and S_0 be the size of the program with and without the given optimization, respectively. $R = S_0 - S$ represents the reduction achieved by such optimization. Let T be the running time of the optimization pass. We define the *average reduction speed* as:

$$ARS = \frac{R}{T}$$

4.2 Code Size Reduction

Figures 5 reports the reduction on the size of the linked object files produced by the compiler. While limiting alignment at a basic block granularity seems restrictive, its effect on code size is small. Even the worst performing variants of HyFM are still within 3 percentage points of SalSSA, while both [PA] and [NW] achieve good results that are on a par with SalSSA. [PA]'s code size reduction varies from 5 percentage points worse to over 10 points better than SalSSA. On average, it is within 1 percentage point of the reduction achieved by SalSSA. [NW] almost always achieves better code size reduction than [PA] and on average outperforms SalSSA by a small margin. Since our primary aim is to reduce the high compile-time overheads of SalSSA a small loss of code reduction is acceptable.

[1] All training, testing, and use of the SPEC2006 and SPEC2017 datasets was conducted at University of Edinburgh.

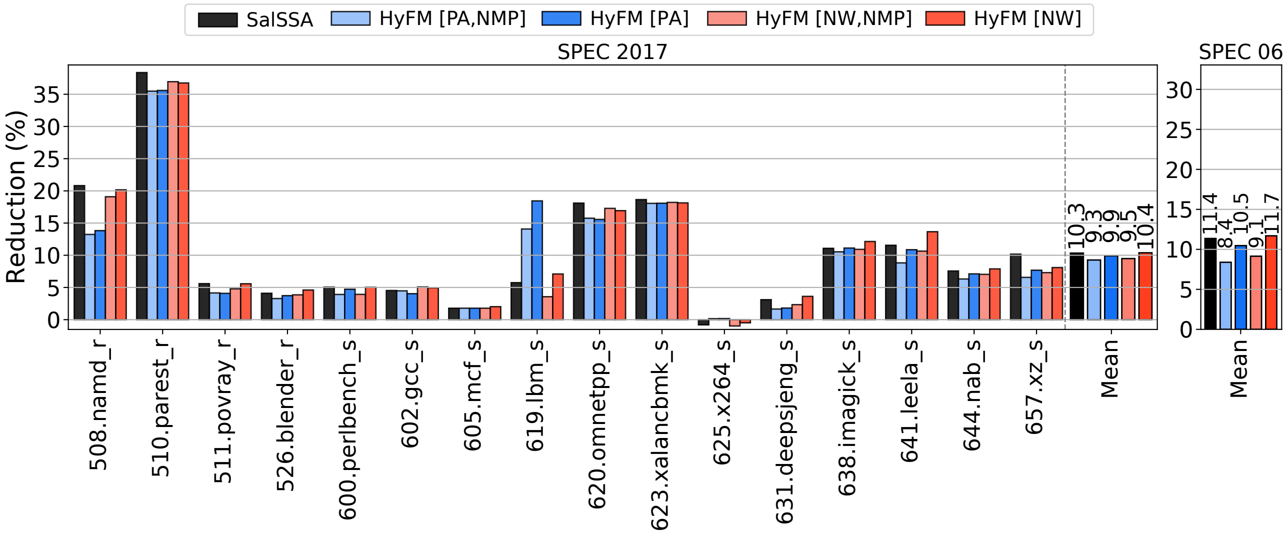


Figure 5. Linked object size reduction over LLVM LTO when performing function merging with HyFM or SalSSA on SPEC CPU 2006 and 2017. On average, HyFM improves code size reduction.

These results indicate that the multi-tier profitability analysis is the single most important component of our approach. The two variants without the multi-tier profitability analysis, [PA,NMP] and [NW,NMP], are consistently worse than their counterparts that include this analysis, i.e. [PA] and [NW]. The multi-tier analysis contributes on average about 1 percentage point in code reduction for SPEC 2017 and more than 2 points for SPEC 2006. The multi-tier profitability analysis has an important impact in the quality of the merged function. While SalSSA lets unprofitable merged subsequences through as long as they are outweighed by profitable subsequences elsewhere in the merged function, HyFM filters such unprofitable subsequences out.

The next most important effect comes from the choice of alignment algorithm. Needleman-Wunsch is on average half a percentage point better than Pairwise Alignment for SPEC 2017 and about one percentage point better for SPEC 2006. Given that Pairwise Alignment only aligns blocks of the same size and does not try to discover optimal alignments, this difference is smaller than one would expect. It indicates that profitable pairs of basic blocks tend to be extremely similar if not identical, as discussed in Section 2. Still, Needleman-Wunsch results in more size reduction. When code size reduction is paramount, [NW] might be a better choice than [PA], but as we will see in Sections 4.4 and 4.6, there is still a trade-off to navigate.

We get the biggest improvement by [PA] compared to SalSSA and [NW] for `lbm`, where it reduces the program's object file by 18.5%, almost 13 percentage points more than the competition. This represents an overall reduction of almost 4.2 KB from an object file with a total of 21 KB. SalSSA is able to profitably merge two pairs of functions. On the other hand, [PA,NMP] chooses to perform a chained merge of the three largest functions in `lbm`, resulting in a significantly smaller binary. This is possible because [PA,NMP] is merging some nearly identical pairs of basic blocks of the same size. With the multi-tier profitability analysis, [PA] successfully identifies all four cases. [NW] fails to identify all of these cases, even though it is still better than SalSSA. This exposes existing limitations in the cost model used by our profitability analysis. 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 [19–21, 23].

The two worst results for [PA] are for the `namd` benchmark in both SPEC 2006 and SPEC 2017. SalSSA achieves close to 7 percentage points more than [PA] in code size reduction. In both cases, Pairwise Alignment limits the number of successful merge operations. The variants using Needleman-Wunsch recover most of the lost reduction.

4.3 Speeding Up Function Merging

Figure 6 shows the speedup of HyFM relative to SalSSA. This considers only the time taken by the function merging pass, which include all stages discussed in Section 2.2. Our novel technique achieves an impressive speedup. For [PA] it is on average 5.28× faster for SPEC 2017 and 4.59× for SPEC 2006. Even in the worst case, it achieves a 50% speedup. In the best case, for the SPEC 2017 `gcc`, function merging under HyFM takes a total of 23.5 seconds instead of 302, which translates to almost 13× less time.

All components of HyFM contribute towards this result but the multi-tier profitability analysis has the most significant impact. The two variants with the multi-tier profitability analysis achieve on average three to four times higher

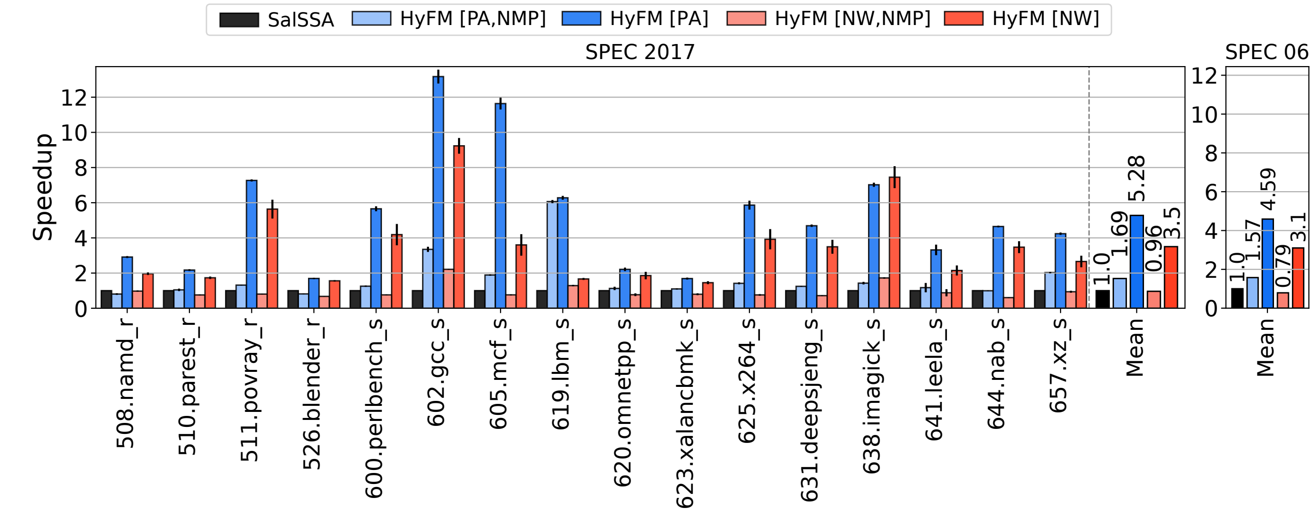


Figure 6. Speedup of the function merging pass in isolation relative to SalSSA. The multi-tier profitability analysis reduces the number of unprofitable merge operations leading to a significant speedup.

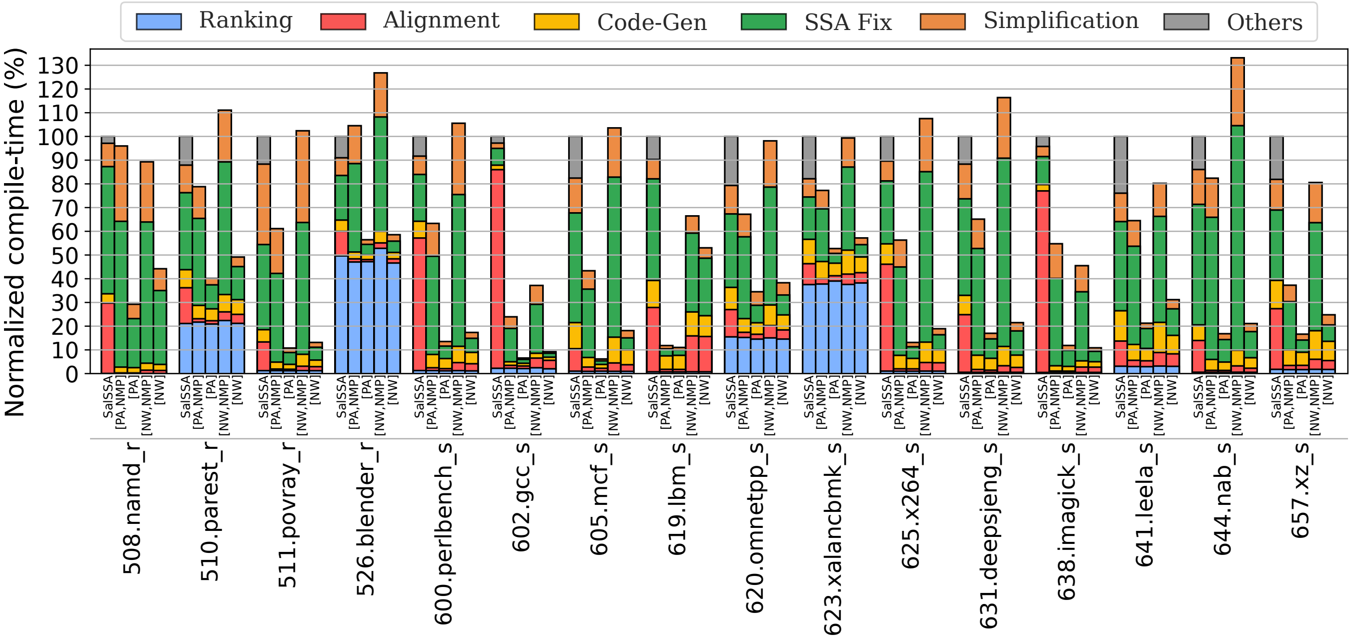


Figure 7. Breakdown of the relative runtime for the different stages of the function merging pass. All measurements are normlized by SalSSA's total runtime on the corresponding benchmark. For every benchmark, we show SalSSA, [PA,NMP], [PA], [NW,NMP], and [NW], in this order.

speedups than their counterparts without it. To help us understand why, Figure 7 shows how the compilation time of each approach is distributed across its various stages. Even though the time spent on the alignment strategy becomes negligible with HyFM, the less optimal alignment often produces complex merged functions — code with an excessive amount of branches, phi-nodes, and operand selections — slowing down SSA reconstruction and code simplification. This effect is very pronounced for the `blender` benchmark, where both [PA,NMP] and [NW,NMP] are slower than SalSSA due to the added pressure on the SSA reconstruction algorithm, even though the alignment overhead is practically zero. Similar effects can be observed in other benchmarks.

Enabling the multi-tier profitability analysis counters this effect by focusing code generation exclusively on profitable blocks and functions. Most of the complex basic blocks HyFM

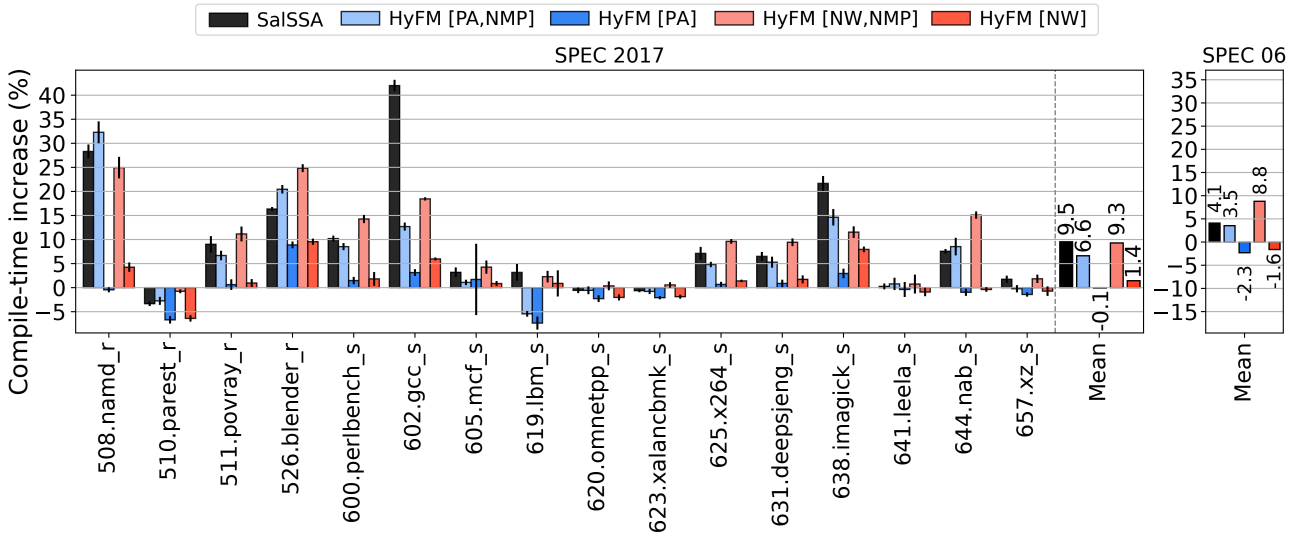


Figure 8. Normalized end-to-end compilation time for SPEC 2017 and SPEC 2006 relative to LLVM LTO.

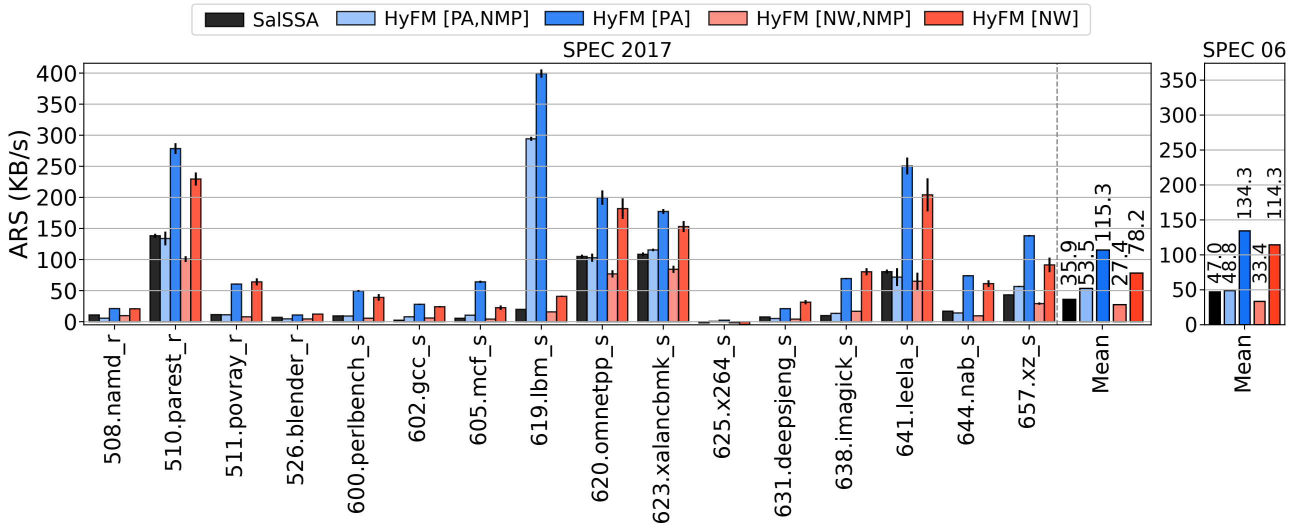


Figure 9. Average reduction speed on both SPEC 2006 and 2017.

generates are not profitable for the same reason it is expensive to process them. The first-tier profitability filters them out. On top of that, most paired functions under either SalSSA or HyFM are unprofitable. SalSSA has to merge them anyway to determine their profitability. This is expensive and wasteful. Our approach, on the other hand, is able to estimate profitability early. Only function pairs with any chance of being profitable, that is pairs with at least one profitable pair of basic blocks, move forward to the expensive merge stage.

The linear pairwise alignment contributes to the performance improvement, too. The variants using pairwise alignment run on average 48% to 98% faster than their Needleman-Wunsch counterparts. The most pronounced case is for `lbm` where [PA] is around 3× faster than [NW]. The blocks paired in `lbm` are longer than usual, so the quadratic Needleman-Wunsch spends significantly more time trying to align them than our linear pairwise algorithm. Figure 7 shows that the added pairing restrictions from [PA], to focus on blocks with higher similarities, also benefits later stages.

4.4 End-to-End Compilation Time

We have also analyzed separately the end-to-end compilation time because reducing code size through function merging has knock-on effects in later stages of the compilation pipeline. The first order effect is that reducing the number of functions tends to reduce compilation time. This is not guaranteed though, because merged functions may be more complex, potentially slowing down later compiler analyses and transformations. Moreover, the time spent merging functions may be so large that it negates any benefits from having fewer functions later in the pipeline.

Even though on a few occasions SalSSA reduces end-to-end compilation time, in general, its overhead is large enough to result in an overall compilation time slowdown, 9.5% to 4.1% for SPEC 2017 and 2006 respectively. In contrast, our approach is so much faster that its compilation time overhead is matched or outweighed by the speedup in later stages. This reduction is marginal for SPEC 2017, but for SPEC 2006 [PA] reduces the average compilation time by 2.3% and [NW] by 1.6%. There is only a single case where [PA] results in a significant end-to-end slowdown, 10% for `blender`. Figure 7 shows that, although [PA] runs faster than SalSSA, both of them spent a significant amount of time ranking the function candidates, due to its large number of functions. Ranking alone in this case takes around 70 seconds.

Overall, we believe that this reduction in end-to-end compilation time is a very important result. While HyFM is still achieving code-size reduction on par with the state-of-the-art, the end-to-end compilation time becomes often faster than the alternative without any function merging. These results suggest that our optimization could always be enabled when optimizing for code size since it is beneficial both in terms of code size and compilation time.

4.5 Code Size and Compilation Time Trade-Off

While the multi-tier profitability analysis improves both code-size reduction and compilation speed, the choice of alignment algorithm introduces a trade-off. Pairwise alignment is better for speed, Needleman-Wunsch is better for code-size reduction. In terms of compilation efficiency, i.e. how much code size reduction we get for the effort we put in, the picture is clearer. In Figure 9, the *average reduction speed* suggests that [PA] achieves the ideal trade-off, with an average reduction speed of 115.3 KB/s, which is around 3× greater than SalSSA's and 20% to 40% greater than [NW].

4.6 Memory Usage

Another important aspect of function merging is peak memory usage. This is especially critical for an optimization designed for LTO. Compilation in full LTO mode is already memory hungry. Just keeping the whole program in memory can be a significant problem for large programs [10]. Maintaining additional information for every function and basic block could easily tip the compiler over the edge.

Figure 10 shows the peak memory usage (in log scale) needed for the alignment stage alone. For SalSSA, this represents s imply the execution of the Needleman-Wunsch algorithm. For HyFM, the alignment stage represents both aligning each pair of basic blocks as well as the pairing these basic blocks. Our results show that [PA] is over three orders of magnitude better than SalSSA, while [NW] is more than two orders of magnitude better. In other words, while SalSSA requires on average 2.4 GB of memory, [PA] uses only around 610 KB and [NW] uses 5.6 MB.

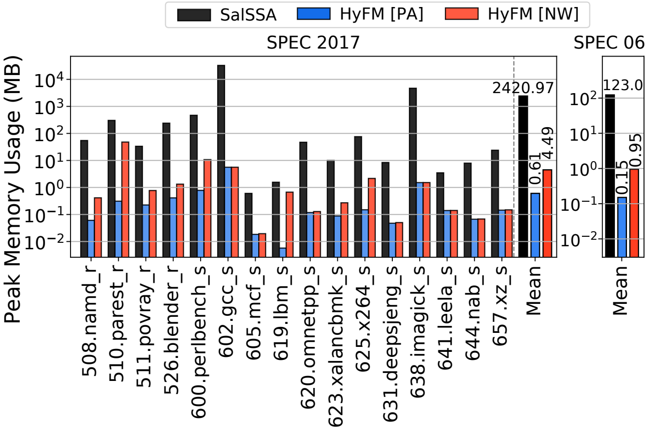


Figure 10. Peak memory usage of SalSSA and HyFM variants for SPEC 2006 and 2017 in log scale. SalSSA has a peak memory usage several orders of magnitude hundreds higher than all other approaches. The pairwise alignment variants of HyFM need on average only a seventh of the memory needed by the Needleman-Wunsch variants.

The peak memory usage is especially noticeable on `gcc`, when merging its two largest functions, containing 90093 and 76265 instructions. Since SalSSA applies its quadratic sequence alignment algorithm on the linearized sequences of the whole input functions, it uses over 32 GB of memory when merging these two functions. Meanwhile, [NW] requires only around 5.6 MB for merging the same pair of input functions, even though it employs the same sequence alignment algorithm. This is because its peak memory usage is a quadratic function of the largest pair of *blocks* instead of the largest pair of *functions*. Although very large, these two functions are composed of several thousands of very small basic blocks, so the memory overhead of Needleman-Wunsch is limited. Most of the memory consumed by [NW] in this case is actually needed for storing the basic block fingerprints. This aspect becomes evident when we compare the peak memory usage of [NW] with that of the [PA] for `gcc`. They have similarly low memory requirements, even though only one of them uses a quadratic alignment algorithm.

In other cases, where basic blocks are longer, pairwise alignment leads to a significantly lower peak memory usage compared to Needleman-Wunsch. For `parest`, for example, pairwise alignment reduces memory usage from 40 MB to 200 KB. Overall, [PA] needs around 6× less memory. For smaller programs, [NW] might be a viable option but for larger ones being able to reduce memory usage to a minimum might be more important.

4.7 Summary

Overall, our novel function merging technique has surpassed the state-of-the-art in terms of compilation time, memory usage, as well as code size reduction. However, different variants of the proposed technique are better suited for different goals. If the code size is the utmost concern, HyFM [NW] is the winning strategy, but if we are looking for the most balanced trade-off between compilation-time overheads and code-size reduction, HyFM [PA] has shown better results.

5 Related Work

Compiler-based code size reduction is important for fitting large programs to resource-constraint embedding devices. Previous approaches reduce code size by replacing a code segment with a smaller, semantically-equivalent implementation [17, 29], deleting unnecessary code [7, 12], combining redundant code within a function [5, 6] or across functions [4, 16]. Function merging falls into the later category.

Established compilers like GCC and LLVM [2, 15] provide an optimization for merging identical functions at the IR level. They can only handle type mismatches that can be losslessly cast to the same format. Von Koch et al. [9] extended this idea into merging nearly identical functions. They restrict merging to functions with the same signature, identical control-flow graphs, corresponding basic blocks must also have the same number of instructions, and corresponding instructions in these basic blocks must have the equivalent data type. They only allow pairs of corresponding instructions to differ in their opcode or list of arguments.

Rocha et al. [23] proposed a technique capable of merging arbitrary pairs of functions. They employ a sequence alignment algorithm to find equivalent code segments. These aligned functions with equivalent code can be merged into a single function. Mismatching code segments of code are also added to the merged function but have their code guarded by a function identifier. SalSSA [24], based on the sequence alignment algorithm, is the current state-of-the-art function merging technique. While promising, SalSSA still suffers from high memory usage and compilation time. As we have shown in the paper, when compiling a modest-sized program, the memory requirement of SalSSA can go well beyond what is typically available to a developer. This drawback limits the practicability and the scale at which SalSSA can operate. HyFM overcomes the limitations of SalSSA with a novel alignment strategy and probability analysis. Experimental results show that our techniques significantly reduce the memory consumption and compilation time required when using the sequence alignment algorithm, allowing function merging to scale to larger programs.

Link-time optimization can merge text-identical functions at the machine instruction level [1, 13, 28]. This technique is hardware-specific but is orthogonal to our techniques that work at the compiler IR level. Another closely related technique is procedural abstraction [4, 8, 16]. This technique moves identical code segments to separate functions and replaces the original code segment with a function call. It requires the code texts to be fully identical, but our function merging approach does not have this restriction.

6 Conclusion

We have presented HyFM, a novel technique for compiler-based function merging. By operating on individual pairs of basic blocks, it eliminates most of the time and space overheads of the previous state-of-the-art approach. Through its multi-tier profitability analysis, it allows the compiler to bail out early from unprofitable merging attempts saving additional compilation time.

We evaluate HyFM by applying it to SPEC CPU2006 and 2017 benchmark suites. Experimental results show that HyFM achieves comparable and often better results in code size reduction than the state-of-the-art. However, HyFM achieves this while running over 4× faster and using orders of magnitude less memory. We further demonstrate how different variants of HyFM can be developed, giving users the flexibility to control the trade-off between compilation overhead and code size reduction.

Future work for improving function merging could focus on an even finer grain tier to the profitability analysis that works on individual pairs of aligned instructions. Another aspect that could be improved is the strategy for pairing functions, the most expensive part of function merging for some benchmarks. One could also explore a hybrid approach that uses both the pairwise and the Needleman-Wunsch alignment algorithms, depending on the size of the paired basic blocks.

Acknowledgment

This work has been supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grants EP/L01503X/1 (CDT in Pervasive Parallelism), EP/P003915/1 (SUMMER) and EP/M01567X/1 (SANDeRs). This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

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

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

[3] Rafael Auler, Carlos Eduardo Millani, Alexandre Brisighello, Alisson Linhares, and Edson Borin. 2017. Handling IoT platform heterogeneity with COISA, a compact OpenISA virtual platform. *Concurrency and Computation: Practice and Experience* 29, 22 (2017), e3932. https://doi.org/10.1002/cpe.3932

[4] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-size Optimization for Production iOS Mobile Applications. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, US, 1–12. https://doi.org/10.1109/CGO51591.2021.9370306

[5] Wen Ke Chen, Bengu Li, and Rajiv Gupta. 2003. Code Compaction of Matching Single-Entry Multiple-Exit Regions. In *Static Analysis*,

Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–417. https://doi.org/10.1007/3-540-44898-5_23

[6] John Cocke. 1970. Global Common Subexpression Elimination. In *Proceedings of a Symposium on Compiler Optimization*. ACM, New York, NY, USA, 20–24. https://doi.org/10.1145/800028.808480

[7] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, USA) *(LCTES '99)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/315253.314414

[8] A. Dreweke, M. Worlein, I. Fischer, D. Schell, T. Meinl, and M. Philippsen. 2007. Graph-Based Procedural Abstraction. In *International Symposium on Code Generation and Optimization (CGO'07)*. 259–270. https://doi.org/10.1109/CGO.2007.14

[9] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. 2014. 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)*. ACM, New York, NY, USA, 85–94. https://doi.org/10.1145/2666357.2597811

[10] Teresa Johnson, Mehdi Amini, and Xinliang David Li. 2017. ThinLTO: Scalable and Incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 111–121. https://doi.org/10.1109/CGO.2017.7863733

[11] S. L. Keoh, S. S. Kumar, and H. Tschofenig. 2014. Securing the Internet of Things: A Standardization Perspective. *IEEE Internet of Things Journal* 1, 3 (June 2014), 265–275. https://doi.org/10.1109/JIOT.2014.2323395

[12] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial Dead Code Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 147–158. https://doi.org/10.1145/773473.178256

[13] Doug Kwan, Jing Yu, and B. Janakiraman. 2012. Google's C/C++ toolchain for smart handheld devices. In *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*. 1–4. https://doi.org/10.1109/VLSI-DAT.2012.6212583

[14] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: Inter-Procedural Basic Block Layout Optimization. In *Proceedings of the 28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. Association for Computing Machinery, New York, NY, USA, 65–75. https://doi.org/10.1145/3302516.3307358

[15] Martin Liška. 2014. Optimizing large applications. *arXiv preprint arXiv:1403.6997* (2014).

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

[17] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36177.36194

[18] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443 – 453. https://doi.org/10.1016/0022-2836(70)90057-4

[19] A. Pohl, B. Cosenza, and B. Juurlink. 2018. Cost Modelling for Vectorization on ARM. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. 644–645.

[20] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 206–216. https://doi.org/10.1109/CGO.2019.8661192

[21] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. Look-ahead SLP: Auto-vectorization in the Presence of Commutative Operations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/3168807

[22] Rodrigo Rocha. 2020. pldi20salssa. (4 2020). https://doi.org/10.6084/m9.figshare.12089217.v1

[23] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 149–163. https://doi.org/10.1109/CGO.2019.8661174

[24] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 854–868. https://doi.org/10.1145/3385412.3386030

[25] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen, and Jørgen Lindskov Knudsen. 2003. Compiling Java for Low-end Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems* (San Diego, California, USA) *(LCTES '03)*. ACM, New York, NY, USA, 42–50. https://doi.org/10.1145/780732.780739

[26] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder. 2012. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine* 50, 12 (December 2012), 144–149. https://doi.org/10.1109/MCOM.2012.6384464

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

[28] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit*.

[29] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. 1982. Using Peephole Optimization on Intermediate Code. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 21–36. https://doi.org/10.1145/357153.357155

[30] A. Varma and S. S. Bhattacharyya. 2004. Java-through-C compilation: an enabling technology for Java in embedded systems. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 3. 161–166 Vol.3. https://doi.org/10.1109/DATE.2004.1269224