Accelerating Function Merging

Sean Stirling - s1641210

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

The importance of file compression in hardware limited devices has long been known; compressing files can free up valuable storage and memory space on SoC devices, smartphones, personal computers and web servers. In 2019, a new technique paved the way for binary file compression through an additional step in compilation by merging similar functions, borrowing a technique from BioInformatics known as sequence alignment. In its infancy and later upgrades, this technique provides far greater code reduction than its counterparts but falls short due to its increased compile-time requirements.

In this paper, we present a vastly superior search strategy that achieves improvements in code reduction, memory usage, and compilation times when compared to the current state-of-the-art. Applied to the SPEC 2006 suite, we show that our approach delivers average compilation times below that of a baseline which performs no function merging. Consequently, we show that function merging can serve as a notable speedup in compilation by reducing the effort of the code generator.

Acknowledgements

Huge thanks to Pavlos Petoumenos and Rodrigo Rocha for supervising me throughout the development of this project. Their guidance, feedback, and support has been monumental to the success of this thesis. It has been a privilege to be supervised by them.

Table of Contents

Chapter 1

Introduction

1.1 Project Description

The demand for the consideration of resource-constrained devices is becoming more and more apparent in recent years with the rapid development of the Internet of Things. Such systems are very limited in their memory and storage capacities. When it comes to compiled code, the sequence alignment technique for merging functions can greatly alleviate this issue by combining function bodies such that the resulting merged function contains the instructions for both functions but de-duplicates the shared ones. What comes out is a merged function that requires less space but has behaviour identical to that of the original functions.

This technique relies on a series of steps to format, identify, align and merge functions. Last year, the most computationally expensive part of this process, sequence alignment, was investigated to reduce total compilation times and increase code reduction. In this paper, a broader range of steps in this process are explored so as to further reduce compilation overhead and increase the code reduction capabilities of the technique. More specifically, this paper investigates the search strategy used to identify similar functions and briefly visits the system used to determine the profitability of merging two functions.

1.2 Motivation

File compression cleverly packs data into a format such that it can be read and understood just as well as the original file but at a much smaller size. A compressed file will take up less system storage and memory, which becomes incredibly important for hardware-constrained devices. With most files, simply executing a compression algorithm onto them can produce an acceptably compressed file. However, for generated binaries, these methods cannot be applied without breaking the program.

One solution, executable compression, combines the compressed data with decompression code into a single executable. On execution of this file, the decompression code recreates the original code before executing it. While successful in its goal to reduce binary file sizes, it comes with a plethora of significant disadvantages. These include: being prohibited by some virus scanners; upon startup of a compressed exe/dll, all of the code is decompressed from the disk into memory in one pass, which is inefficient on systems already low on memory; and multiple instances of a compressed exe/dll creates multiple instances in memory, whereas, with uncompressed exe/dlls, the code is stored once and shared between instances.

Other techniques to reduce binary file sizes are included on the compiler itself, attempting to shrink the size of compiled code. Unfortunately, these techniques fall flat in their attempts with lacklustre code reduction. Production compilers offer little help beyond dead-code elimination or the merging of identical functions. Even experimental state-of-the-art compilers, which analyse control flow graphs (CFGs) to determine merging possibilities, don't achieve significant code reduction. This is likely caused by the rigid, overly restrictive algorithms used to find candidates.

2019 saw the emergence of a new technique for code reduction by merging the bodies of similar functions using a technique borrowed from BioInformatics known as sequence alignment. This method has significant benefits over other merging techniques, in that it is possible to merge any function with another, whether they have varying CFGs or signatures. With the increased flexibility in merging capabilities, a greater number of functions can be merged reducing the impact of code duplication.

Yet, it disappoints with its increased compilation times. Relying on some inefficient methodologies to achieve the desired merges leaves the technique with unrealized potential. For instance, the exploration framework used to identify pairs of functions to be merged is quadratic at its core. As code-bases grow and the number of functions increases, this framework becomes unable to efficiently find similar functions and becomes a key slowdown in the process. Take the DealII and Xalancbmk benchmarks from the SPEC2006 suite [1], and the LLVM [2] code-base: of the whole function merging procedure, the relative time spent ranking functions for similarities totals $\sim$33%, $\sim$69%, and $\sim$98% respectively. Ranking becomes the predominant process the optimisation spends its time on.

Moreover, this framework attempts to merge every single function with another, regardless of how similar those two functions may be, resulting in the overall time being dedicated to wasteful failed merges. This can be clearly seen in the gcc and povray benchmarks, where the proportion of the processing of failed merges to the total optimisation time, totals $\sim$92% and $\sim$90% respectively. For many benchmarks, the system is committing an egregious amount of time to failed merges.

1.3 Objectives

The aim of this project is to improve the effectiveness and performance of the current state-of-the-art in code size optimisation. Through analysis of this system, solutions can be implemented that allow it to become practical for all applications, realising its potential for widespread adoption.

1.4 Contributions and Results Summary

In this paper we present a search strategy based on the MinHash algorithm [3] applied with the Locality Sensitive Hashing technique to achieve rapid similarity searching between functions. Through the SPEC2006 benchmark suite [1] and select large-case programs, we show that our method is superior in code reduction, memory usage, and compilation time, when compared to the current state-of-the-art, SalSSA [4].

Furthermore, we show that our method achieves boosts in compilation time and improvements in memory requirements against a baseline compiler that performs no function merging. This allows us to present function merging as a noteworthy speedup in compilation time while also lowering the required resources to compile programs.

SPEC2006	Baseline	SalSSA	Ours
Avg Code Reduction (%)	N/A	10.2	10.6
Avg Relative Compilation Time (Normalised to the baseline) (%)	N/A	11	-4
Avg Peak Memory Usage (MB)	152	159	146
LLVM			
Code Reduction (%)	N/A	30.2	30
Compilation Time (s)	1011.48	11139.2	859.24
Peak Memory Usage (GB)	9.86	8.34	7.3
Linux Kernel			
Code Reduction (%)	N/A	N/A	N/A
Compilation Time (s)	25.97	1376.34	30.13
Peak Memory Usage (GB)	1.37	1.177	1.184

Table 1.1: Results

Chapter 2

Background Review

2.1 Code Size Optimisation

Optimisation of written code is at the heart of an optimising compiler's purpose; cleverly transforming code during the compilation process to improve specific attributes in the output binaries. With these compilers, programmers can select for minimised execution time, memory requirements, power consumption, or binary size. Systems low on storage and memory hardware will benefit greatly from reduced binary sizes, reducing costs or allowing for hardware to be put to other necessary tasks.

The vast approaches to code size optimisation [5] can be split into two categories: the replacement of code with smaller but semantically equivalent code; and the removal of redundant code. The former employs techniques like peephole optimisation [6], whereas the latter employs techniques such as dead code elimination and function merging. Function merging bases itself on the premise that similar functions should be replaced with a singular function representing the individuals as one. It is common for compilers to merge identical functions as seen by Google's Identical Code Folding (ICF) technique [7] implemented into their ELF gold linker [8] at the bit level, and LLVM's provided identical merging pass.

Prior to sequence alignment, the state-of-the-art [9] merged functions when they share identical structures and signatures. For two function signatures to be equivalent they must agree on the number, order, and types of arguments, as well as other compiler-specific properties. Secondly, the functions must share identical Control Flow Graphs (CFGs). There must exist a directed edge-preserving bijection between them, known as graph isomorphism. In other words, the number of vertices and edges are the same and their edge connectivity is also equivalent. Lastly, the number of basic blocks must be equal, with each instruction having equivalent types. The rigidity of this technique prevents many highly similar functions from being merged and its results on the SPEC2006 benchmark suite shows this at only an average 3.9% reduction in code size while introducing an extra 5.4% overhead to compilation time.

Sequence alignment, as a technique for code reduction, was published in 2019 as FMSA (Function Merging by Sequence Alignment) [10]. It did not suffer from any of the major limitations of previous solutions, outperforming them by more than 2.4x in terms of code reduction. FMSA achieved an average 6% code reduction on the SPEC2006 benchmark suite but lagged behind other solutions with an added average 15% compile-time requirement.

2.2 LLVM Compiler Architecture

The LLVM project [2] was initially designed to provide a compiler framework that allowed for life-long program analysis and transformation. Since then, LLVM has matured into a collection of modular and reusable compiler and toolchain technologies, cementing LLVM as an umbrella project that encompasses many sub-projects.

The LLVM compiler implements the three-phase multipass compiler architecture, dividing source code processing into three distinct stages. This design is depicted in Figure 2.1, consisting of a frontend, an optimiser, and a backend. The frontend parses and validates source code, transforming it into an Intermediate Representation (IR) and passes that to the optimiser. The optimiser uses the IR to transform the source code, independently of the source language it was written in, to improve aspects of the code such as code size, execution speed, or memory usage. Function merging sits as a stage within the optimiser. The backend, or code generator, finally transforms the IR into the target machine code.

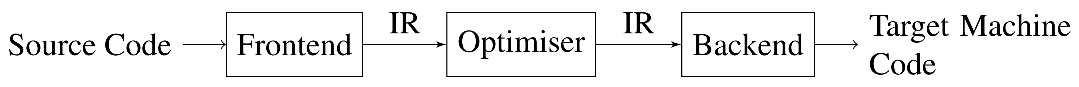


Figure 2.1: Simplified LLVM Compiler Architecture

2.2.1 LLVM Intermediate Representation

At the core of the LLVM project is its IR. A strongly typed, assembly-like, language that abstracts away the specifics of the higher-level source language and the details of the target. This allows programmers to develop their compiler optimisations to work on any language, such that it has a frontend to transform to IR. An example LLVM IR instruction can be seen in Figure 2.2.

Variables are distinguished by the % symbol, as seen by %8 and %7. An instruction's opcode simply defines what the instruction is doing. In the case of Figure 2.2, it is adding the variable %7 and the number 1 together. The resulting type of the instruction is given right next to the opcode, an i32 (32-bit integer) in this case.

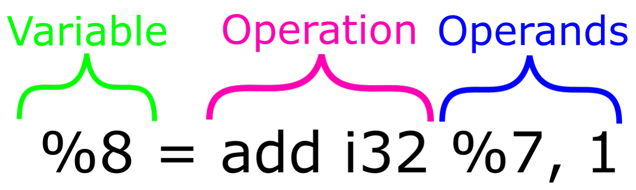


Figure 2.2: Example IR Instruction

Function merging operates by matching the individual equivalent instructions within each function. In general, for two IR instructions to be equivalent they must follow these set of rules:

1. Their opcodes must be equivalent
2. The instruction types must be equivalent
3. They each contain the same number of operands
4. Each pairwise operand must have equivalent types

This, however, is only a generalised idea of what makes two instructions equal. For many instructions, they must be examined further at an increased depth to determine whether they are equal or not, while some cases can be more lenient on these rules. For instance, the equivalence of Call instructions depends on the equivalence of the called functions, whereas Return instructions can be equivalent regardless of the number of operands.

2.2.2 Static Single Assignment

An important aspect of LLVM's IR is that it adheres to the Static Single Assignment (SSA) form, meaning that variables are assigned only once and defined before they are used. This strict rule where each variable is defined prior to their uses is known as dominance. Existing variables within the source code which do not adhere to these rules are split into new variables known as *versions*, usually indicated by subscript notation e.g x_1 and x_2. SSA form is responsible for simplifying and enhancing a number of compiler optimisations since the necessary use-def chains (uses and definitions of a variable) are explicit and do not need to be calculated.

Most importantly for this paper, are the existence of phi-nodes. Due to the demands of the SSA form, the problem arises where multiple definitions come from separate branches of execution. For instance, consider the leftmost CFG in Figure 2.3.

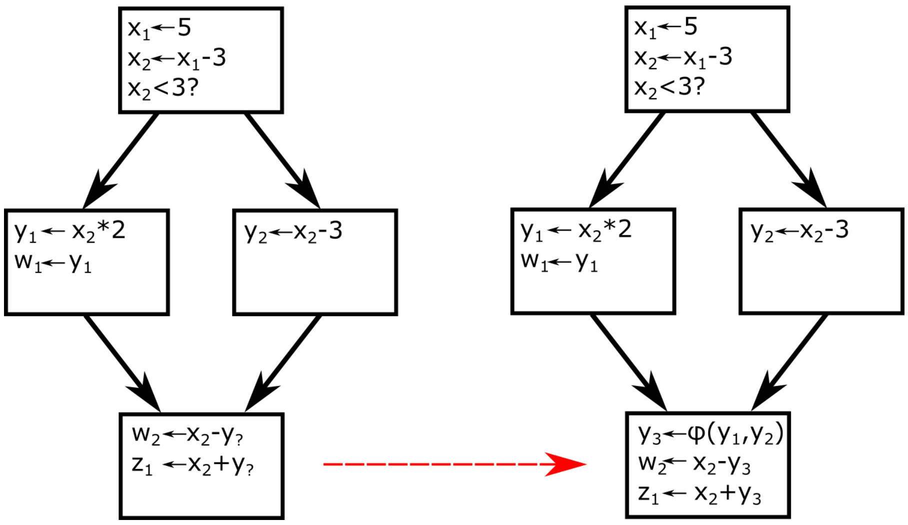


Figure 2.3: Phi-Nodes - Resolving Definitions

The final basic block on the left is unable to resolve which definition *y* should assume. The correct definition of the variable, which is used in subsequent instructions, could come from either incoming branch. To resolve this, a special instruction known as a phi-node, ϕ, is inserted at the entrance of the uniting basic block. This phi-node selects for which incoming definition between the branches and can resolve more than just two definitions.

Phi-nodes are not directly implemented as machine operations. In LLVM's case, when compilation is near completion, SSA form is eliminated. In most cases, phi-nodes should not present an increase in code size as they are not actually translated to machine instructions. However, during the register allocation stage, where variables are assigned onto the small number of CPU registers, phi-nodes can exacerbate *register pressure* and cause an increase in load and store instructions. If there are not enough registers available to hold all the necessary variables, some will be moved to and from memory, known as *spilling* the registers. Code which accesses the registers can be more compact than instructions fetching and storing in memory.

2.3 SalSSA

In less than a year since the initial release of FMSA, the sequence alignment technique has been upgraded, seeing far greater code reduction and even better compilation times. This new system, SalSSA [4], geometrically averages 9.3% in code reduction while minimising compilation overhead to just 5%.

SalSSA improves upon the FMSA system with a major overhaul in the way it handles phi-nodes in the SSA form. FMSA demanded the removal of these instructions before merging and replaced them with costly extra code, then added them back in once the function merging step was complete, which unfortunately brought increased compilation times and lacking code reduction. Moreover, as function sizes and complexities grew, the negative effects of this method escalated seeing the doubling in size of many merged functions.

Instead, SalSSA efficiently handles these instructions by disallowing one to be merged with another, and by generating new ones when necessary. SalSSA treats phi-nodes as though they are attached to the basic block's label, i.e. they are not used in the computation of the alignment but are copied into the newly merged function in their associated basic blocks. Then, due to the increased complexity of the function, additional phi-nodes are generated to maintain the dominance property (SSA) for these violating variables.

While we present this work on the SalSSA system, it is not exactly equivalent to SalSSA [4] which was released last year. There have been some minor adjustments made which can result in differing code reduction and compilation times compared to that system. We present our work on this slightly modified system. These modifications are unimportant for this paper's discussion.

2.3.1 Component Steps

The SalSSA compiler can be thought of as the general C++ Clang compiler but with the added function merging procedure placed between other optimisations built with LLVM. The SalSSA function merging pass comprises of multiple key component steps:

Searching Before merging two functions, they must first be identified and paired up. Large-scale similarity computations can be very expensive, so the current exploration framework reduces this effort by breaking down functions into fingerprints based on opcodes and types. A function is then merged with its most resembling fingerprint counterpart.

Linearisation Sequence alignment operates on sequences, not CFGs. Prior to aligning two functions, they are flattened into a sequence of instructions by the linearisation algorithm.

Sequence Alignment Finding the most optimal arrangement of two functions into one is the job of the sequence alignment algorithm. By minimising the Levenshtein edit distance [11], or by maximising a score based on an arbitrary scoring scheme, the alignment details how the merged function should look. Equivalent instructions are matched, whereas differences are separated with gaps.

Code Generation With the alignment completed, the final merged function is built from that information. Branches are created to separate the differences between functions and a function identifier is used to select between them depending on which original function was intended to be called.

Cost Profitability Before storing the newly merged function, the system must check that keeping it will give a lower total binary size. One IR instruction does not necessarily translate to one machine instruction, and so the estimation of a function's size is measured with LLVM's target-specific code size cost model.

This general procedure is shown in Figure 2.4, where two functions have been paired up, linearised, aligned, generated, and checked for profitability, resulting in a successful merge.

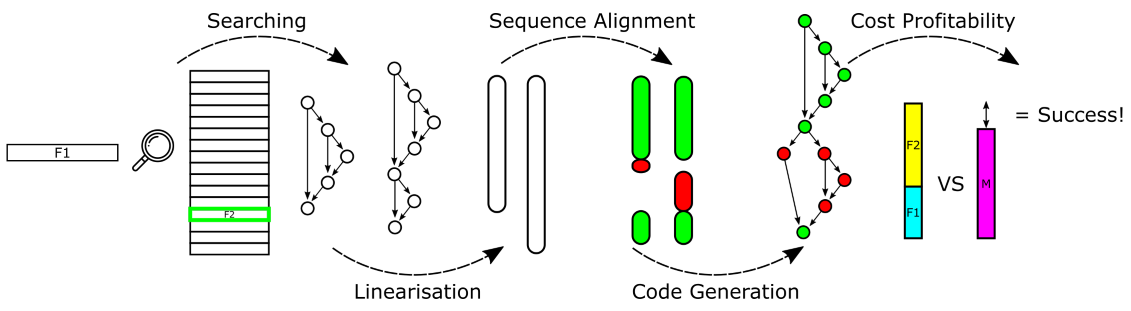


Figure 2.4: SalSSA Overview

2.4 Sequence Alignment

Sequence alignment is one of the most crucial optimisation problems to exist within the field of BioInformatics. It is firmly tied to a similarity measure between sequences, first defined as the edit distance between two strings, known as the Levenshtein distance [11]. This distance represents the minimum number of edit operations - insertions, deletions, and letter substitutions - needed to transform one string into another.

A sequence alignment algorithm solves for a generalised version of this minimum distance and traces through the optimisation space to find the alignment that assumes it. Principally, an optimal alignment represents the closest arrangement of two sequences such that the number of matching characters are maximised while the number of mismatches or gaps are minimised. An example alignment can be seen in Figure 2.5.

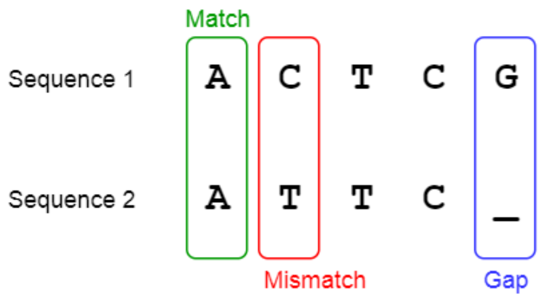


Figure 2.5: Sequence Alignment

Optimally pairwise aligning two sequences requires the effort of dynamic programming solutions at the cost of $\mathcal{O}(NM)$ in time and memory, as seen in Needleman-Wunsch's [12] and Smith-Waterman's [13] algorithms, where N and M are the respective sizes of the two sequences. In fact, it was the Needleman-Wunsch algorithm that was presented in FMSA [10] and used again in SalSSA [4]. Fortunately, the memory requirements can be reduced to the linear memory requirement of $\mathcal{O}(min\{N,M\})$ with Hirschberg's algorithm [14].

Last year's investigation into the sequence alignment technique provided a variety of new algorithms shown to increase code reduction by around 1.5% while introducing no extra compile-time overhead or speed up compilation by 5% at no cost to code reduction.

The various algorithms showed that focusing on only highly similar segments while gapping the rest, or constraining the alignment to keep the number of gap openings quite low, resulted in boosted code reduction. Knowing that gap openings in an alignment indicated branches, and subsequent surplus code, it was concluded that minimising their number plays a significant role in reducing the final size of the merged function, even at the cost of turning down some number of matching instructions. These algorithms included the Smith-Waterman [13], Gotoh [15], and its linear memory counterpart, Myers-Miller [16].

Heuristic algorithms like BLAST (modified to a pairwise algorithm) [17], MUMmer [18], and FOGSAA [19], were implemented to reduce the total amount of time spent aligning. Unfortunately, FOGSAA brought disastrously larger alignment times despite the effort put into its implementation. Unlike FOGSAA, BLAST and MUMmer reduced the total compilation time by 5% and 3% respectively.

2.5 Set and Sequence Similarity

As briefly mentioned, the Levenshtein distance measures how similar two sequences are. While it is commonly used interchangeably with edit distance, the edit distance actually describes a family of sequence similarity measurers including the Hamming distance [20] and Longest Common Substring (LCS) distance. Such similarity measures are also known as string metrics which more broadly attempt to describe the similarities between two sequences. String metrics are incredibly important in the realm of approximate string matching and searching.

String metrics don't just make use of edit distances - with insertions, deletions and substitutions - instead also utilising set similarity measures to give their scores. These include the Overlap Coefficient, Sørensen–Dice coefficient [21], and the popular Jaccard Index [22]. Set similarities like these are applied to sequences by decomposing the sequence into a set of overlapping fixed-size subsequences. For instance, consider the sequence and its resulting subsequences in Figure 2.6. The sequence "ACTCG" has been transformed into a set containing the elements "ACT", "CTC", "TCG".

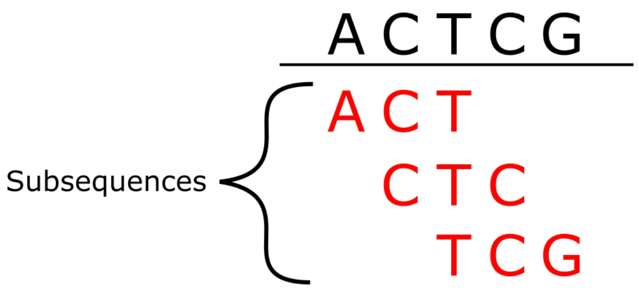


Figure 2.6: Breaking Down a Sequence

Such subsequences go by many different names depending on the field they're being used in. For instance, BioInformatics regularly uses *seeds* or $k-mers$, linguistics may use $n-grams$, and natural language processing may use $w-shingles$. Where k, n, or w, represent the number of characters within the subsequence. We will use the terminology $w-shingles$ for similarity measuring between sets, or just w to represent the size of each subsequence.

2.5.1 Jaccard Index

The Jaccard Index can be calculated with this equation:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

This value is 0 when the two sets are completely disjoint and 1 when they are exactly equal. The Jaccard Index is therefore a similarity measure, not a distance.

Like the edit distances, the Jaccard Index is difficult to compute, this time due to its set operations. Given that we use a C++ set, we could calculate the set operations in $\mathcal{O}(N)$ time, assuming we could sort and hash the elements, where N is the set size. However, in the context of large-scale similarity computation, a linear operation just to compare sequences is out of the question. Comparing hundreds of thousands of variably sized sequences with one another will be dramatically bottlenecked by similarity calculations.

We could even use bloom filters [23] to probabilistically determine whether the two sets share elements or not, within a certain confidence range, at a far superior rate when the sets become large. Utilising random hash functions, bloom filters would allow us to test whether an element belongs to a set in an incredibly memory efficient and rapid constant time. But similarity calculations would still be linear to the number of elements.

2.5.2 MinHash

A solution that addresses this problem is the somewhat esoteric MinHash algorithm [3]. Like Bloom Filters, the MinHash algorithm leverages the randomness in hashing, giving it the ability to probabilistically solve for the Jaccard Index in constant time, $\mathcal{O}(k)$, where k is a constant value. However, it must perform a pre-computation stage that requires $\mathcal{O}(Nk)$ time to complete but does not need to be re-computed. This fact makes it a great technique for large-scale similarity computation.

The elegant solution presented by MinHash involves hashing each element of both sets and selecting the minimum value of each set. Surprisingly, the probability of these two hashes being equal is exactly equivalent to the Jaccard Index of the two sets. The idea, then, is to randomly sample this probability k times to estimate this value. With k samples, the expected error is $\mathcal{O}(1/\sqrt{k})$. For example, 400 samples would estimate the Jaccard Index with an expected error less than or equal to 0.05.

2.5.2.1 MinHash Explained

Suppose we have two sets, A and B. If an element produces the minimum hash in both sets on its own, it also produces the minimum hash in their union.

$$min(h(A)) = min(h(B)) \Leftrightarrow min(h(A \cup B)) = min(h(A)) = min(h(B)) \tag{2.2}$$

The probability that a specific element in their union, $A \cup B$, is selected as the minimum hash is:

$$\frac{1}{|A \cup B|} \tag{2.3}$$

Therefore, the probability that any shared element, $u \in A \cap B$, is selected as the minimum hashed element is equal to:

$$\frac{|A \cap B|}{|A \cup B|} \tag{2.4}$$

Hence, the probability that any element in their intersection is the minimum hashed element, chosen from their union, is equivalent to the Jaccard Index. But recall that if an element produces the minimum hash in their union, then it also produces the minimum hash in both sets on its own:

$$Pr[min(h(A)) = min(h(B))] = \frac{|A \cap B|}{|A \cup B|} \tag{2.5}$$

The Jaccard Index! The probability that equivalent elements from both sets are selected is equal to the Jaccard Index of the two sets. By hashing k times, and counting the number of hashed elements which they share, and that which they don't, the Jaccard Index can be estimated.

2.5.2.2 MinHash Algorithm

To actually estimate the Jaccard Index through MinHash, those k hashes of each set must first be computed. This is done in a pre-processing stage where the hashes are stored for the upcoming similarity calculations. This process is polynomial to the size of the sequences, the number of sequences, and the number of desired hashes per sequence, $\mathcal{O}(NLk)$, with N, L, and k respectively.

This is obviously a very expensive computation and is often considered to be the fundamental computational barrier. Despite being released over 20 years ago, MinHash is still ever-improving and recent developments have shown that it is possible to reduce the need for k passes over the data to generate the hashes, to just one pass [24][25], reducing the complexity to just $\mathcal{O}(NL)$. This variant, known as the single-hash variant, uses one hash function to hash each element in the set and then picks out the smallest k hashes as the full fingerprint.

Despite the chosen variant, once the necessary hashes have been computed for each sequence, MinHash can estimate the Jaccard Index at an incredible speed. The fastest way of doing this is to first sort the hashes and then iterate through each list comparing hashes. Because sorting the hashes allows the similarity calculation to be performed quickly, they are usually sorted during the pre-processing stage and stored in the sorted form. This gives a similarity calculation in $\mathcal{O}(k)$ time which is effectively constant as k is known at compile-time.

2.6 Nearest Neighbour Search

Nearest Neighbour Search is a long-standing optimisation problem that pertains to all walks of computational science including computer vision, recommendation systems, data compression, and plagiarism detection. Given a similarity or distance metric and a data-point, a solution to this problem must be able to efficiently find the closest other data-point.

The simplest solution, linear search, computes the similarity or distances against every other element in the database, saving the closest element found "so far". This is the exact methodology that the SalSSA searching framework uses to compute a function's most suitable candidate for merging. This method is exact, in that it will always return the closest item in every case. However, as databases become large, this type of searching methodology becomes awfully inefficient, revealing the origin of its other name, "naive search".

Other NNS solutions attempt to approximate the closest data-point through using less exact methods. These include graph-based methods such as HNSW [26], specialised methods like projected radial search used on dense 3D maps of geometric data, and hashing methods as seen in Locality Sensitive Hashing.

2.6.1 Locality Sensitive Hashing

The MinHash algorithm doesn't just provide a quick and reliable similarity measure. It brings with it the potential to utilise a technique belonging to the incredible set of techniques known as Locality Sensitive Hashing (LSH). The very simple core concept behind LSH is to hash similar items into the same buckets with high probability. Therefore, with an LSH searching scheme implemented, searching for an item's most similar counterpart would involve hashing that item and comparing against only a select few items which are very likely to be similar.

The LSH techniques can be found anywhere from being used to filter out duplicates in web pages [27], to reducing the dimensionality of high-dimensional data, or to perform lookups of nearby points in a geospatial dataset [28]. They encompass all sorts of similarity measures and methods to hash similar items, through that similarity measure, into the same bucket. While there are many variations on the Locality Sensitive Hashing technique, we will only focus on the MinHash variation.

In the context of MinHash, LSH splits the list of k hashes into *bands* of size *rows*. Each band is a hash of the individual hashes within that band. For instance, the first band is simply a hash of the first *rows* hashes within the fingerprint. With each of these *band* hashes, we can store the reference to the associated set in a hash map. If two sets are similar, then they will likely match at least one of their bands and hash to the same bucket in that hash map. The *bands* and *rows* terminology comes from a representation of the MinHash algorithm using matrices. Further MinHash and Locality Sensitive Hashing information can be found in the fantastic book Mining of Massive Datasets, Chapter 3 [29].

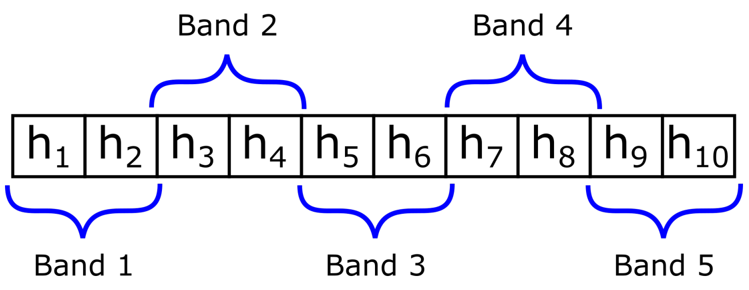

Band 2
Band 4
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10}
Band 1
Band 3
Band 5

Figure 2.7: Decomposing Hashes Into Bands

Consider the list of hashes in Figure 2.7. It can be split into a list of 5 *bands* each made up of two hashes. Using these bands, we can index into a hashmap for each band and compare similarities against the sequences stored in those locations. For example, the 5 *bands* in Figure 2.7 mean we only need to access the hashmap 5 times at those specific band locations and perform full similarity computations against the sequences stored there. Two sequences may not share their first band, and so hash into different locations in the hashmap, missing each other. However, they have 4 more attempts to find each other. If they don't share any bands then they are incredibly unlikely to be similar anyway. It only takes one band match to consider these two sequences as candidate pairs.

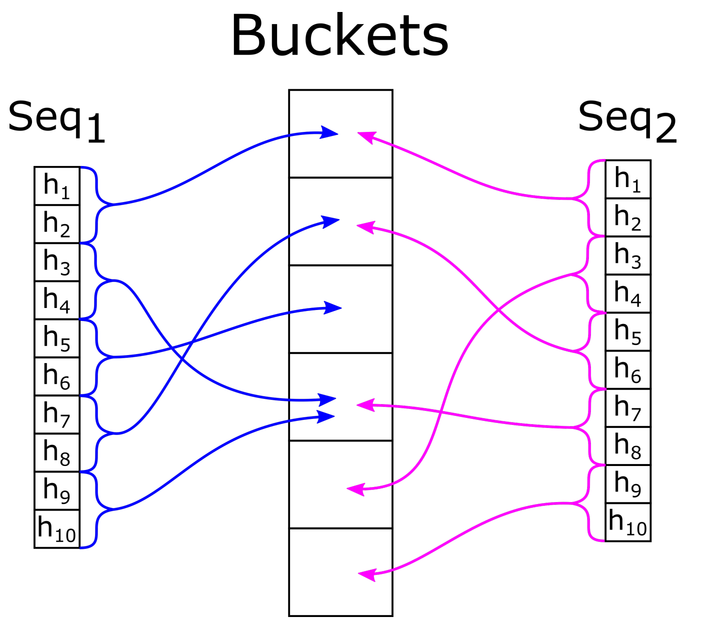

Buckets
Seq_1
Seq_2
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10}
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8 h_9 h_{10}

Figure 2.8: LSH - Similar Sequences Hashing to the Same Bucket

2.6.1.1 LSH Properties

Suppose we use b bands of size r rows, and suppose we have a pair of sets with Jaccard Index s. The probability that these two sets will become a candidate pair can be calculated as follows:

1. The probability that the fingerprints match in every single row is s^r

2. The probability that the fingerprints mismatch in at least one row is $1 - s^r$
3. The probability that the fingerprints mismatch in at least one row of each of the bands is $(1 - s^r)^b$
4. The probability that the fingerprints match in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$

This function takes on the form of an S-curve. Assuming we have 10 hashes per set, setting $b = 5$ and $r = 2$, this curve can be seen in Figure 2.9. As sets become more similar, they become much more likely to become candidate pairs through LSH. As a natural consequence of the probabilities involved within LSH, a threshold is born where sets at an arbitrary similarity are 50% likely to be matched or mismatched. In the case of Figure 2.9, this threshold sits just under a MinHash similarity of 0.4. Pairs above this threshold in similarity are $> 50\%$ likely to become candidates whereas pairs below in similarity are $< 50\%$ likely. This threshold can be approximated with $(\frac{1}{b})^{\frac{1}{r}}$, giving us a threshold of 0.447. If we tune b and r appropriately, we can calibrate the threshold so that we can be sure that function pairs with a certain degree of similarity are very likely to become candidate pairs.

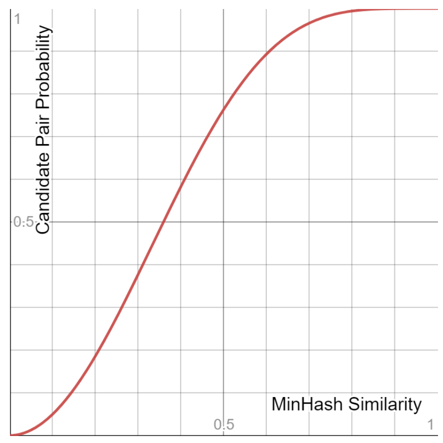


Figure 2.9: LSH - Probability of Becoming a Candidate Pair

Consider Table 2.1, the probability of becoming a candidate pair varies with the similarity between the sets. Very similar documents are guaranteed to become candidate pairs with near 1 probabilities while those that are less similar become far less likely to be paired up.

This would be an ideal scenario if we only wanted to consider comparing against sequences that achieve similarities above 0.7 as we can be more than 96% sure that each sequence will find a specific partner above 0.7 in similarity.

s	$1 - (1 - s^r)^b$
0.1	0.049
0.3	0.375
0.5	0.762
0.7	0.965
0.9	0.999

Table 2.1: Probabilities of Becoming A Candidate Pair at Similarity s. $b = 5$, $r = 2$.

Chapter 3

The Search Strategy

Maximising code reduction with function merging is an incredibly difficult task, one that would require an oracle-like algorithm capable of predicting future merges and allowing functions to be paired up with yet-to-be merged functions. Even constraining to just the currently available functions would still require a quadratic effort per merge just to determine which two functions give the greatest code reduction. Obviously, pairing up functions in this way is completely infeasible and the most attractive solution involves taking each function one-by-one, finding its most similar partner and merging them.

Rapidly measuring similarity between text or sequences is also a difficult task that lands us right back to sequence alignment and the Levenshtein distance [11]. Knowing that such a similarity measure is time consuming to compute, it is far better to strip a function into simple core components that make it very easy to compare with others.

In this chapter, we present a new search strategy that employs the fingerprint methodology through the MinHash algorithm and leverages locality sensitive hashing to rapidly perform searching.

3.1 Previous Search Strategy

On release of the FMSA system [10], it presented a novel exploration framework based on a "light-weight ranking infrastructure" that uses fingerprints to determine similarities between functions. In a pre-computation stage, it calculates a fingerprint for all functions and saves them. A fingerprint consisted of a map of instruction opcodes to their frequency and the set of types in the function. In essence, a function was reduced to the frequency of the operations it performed and the types involved within that function.

The similarity measure between two functions was then calculated with these two equations:

$$UB(f_1, f_2, K) = \frac{\sum_{k \in K} min\{freq(k, f_1), freq(k, f_2)\}}{\sum_{k \in K} freq(k, f_1) + freq(k, f_2)} \tag{3.1}$$

$$s(f_1, f_2) = min\{UB(f1, f2, Opcodes), UB(f1, f2, Types)\} \tag{3.2}$$

Where k in Eq 3.1 is calculating the similarity on opcodes or types. The final similarity measure simply uses the minimum value of these two calculations, as shown in Eq 3.2. Such a score ranged between [0, 0.5], where 0.5 indicated a perfect match, while a 0 indicated that they were completely incompatible.

The exploration framework utilised this similarity measure to linearly scan through the code-base to find the first function's best candidate counterpart, merging and storing the newly created function so that it may be re-merged with others. If this merge was deemed not profitable, it is thrown away and the function is considered to have no suitable merge candidates and is removed from further consideration. This process is repeated for every function until each one has been processed.

This similarity measure and exploration framework remained in the SalSSA [4] system. When it was released in 2020, it touted drastically increased merging possibilities, allowing for more functions to be merged with one another, boosting code reduction all the while reducing compilation times. Unfortunately, the paper fails to mention the success rate of merging two functions. The time spent processing failed merges and successful merges can be found in Figure 3.1. Clearly, the system is struggling to effectively choose when to merge two functions.

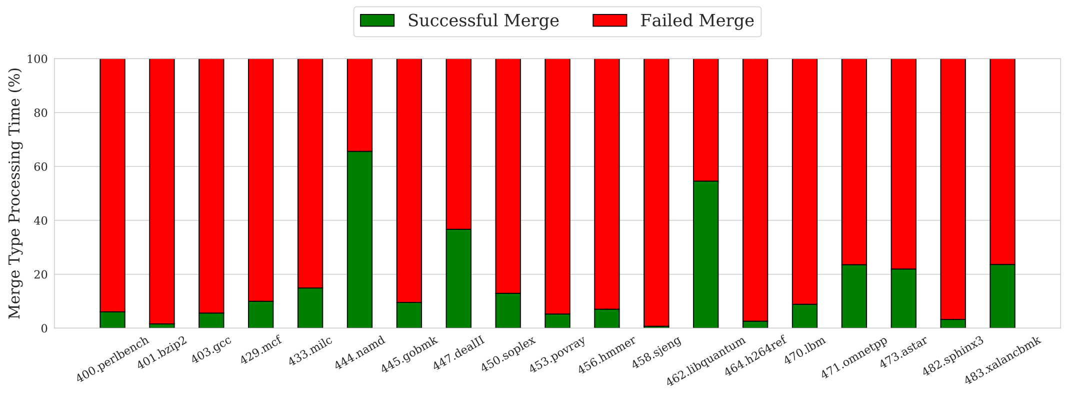


Figure 3.1: SalSSA Processing Times of Successful vs Failed Merges

This problem stems from multiple issues within the search strategy. Firstly, the similarity measure between two functions is unsophisticated. Simply counting the frequency of opcodes and the set of involved types does not fully represent the context to which they belong. Two functions can share similar counts of opcodes and types but can be wildly different in the structure of the blocks which use them.

But by far the worst performance failure of the SalSSA system is that it attempts to merge each function with another. Even if one function is deemed dissimilar to every other function in the code-base, SalSSA will still forcefully try to merge it with the *best* it could find, most often resulting in a failed merge. However, this rigorous methodology guarantees the system superb code reduction by never missing out on any potentially profitable merges.

Despite the time spent on both successful and failed merges, a significant portion of that time is simply spent trying to pair up the two functions, known as ranking. Consider Figure 3.2, quite a few benchmarks exhibit large timings just to rank similar functions.

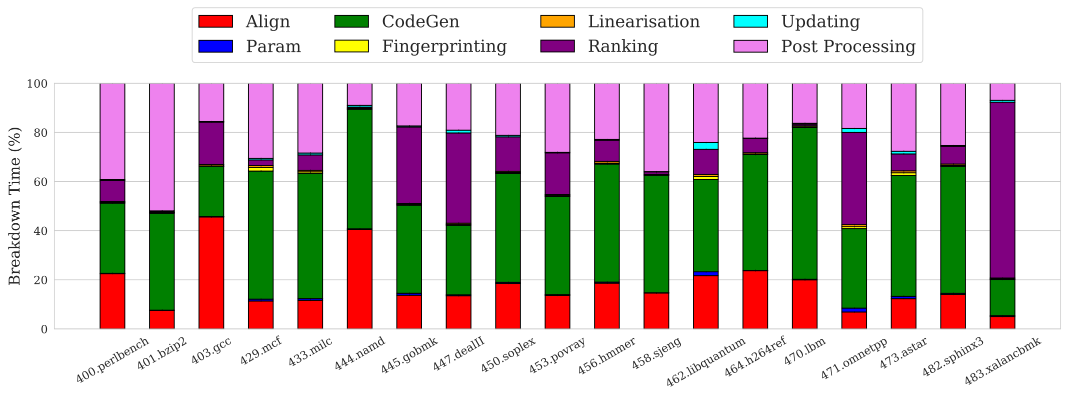


Figure 3.2: Proportion of Time Spent Doing X

This problem gets exponentially worse as we consider real-world programs. The SPEC benchmark suite consists of code-bases that are relatively small compared to what is developed today. For example, attempting to compile the LLVM code-base [2] takes upwards of 3 hours with SalSSA. Because of the sheer size of the application, ~98% of that time is dedicated solely to ranking functions. All other processes within the system are dwarfed by the time spent quadratically ranking over 250,000 functions. This drastically reduces the applicability of the system to real-world programs, preventing it from being adopted as a serious method.

3.2 Overview of the New Search Strategy

To combat these issues, we present a search strategy based on the MinHash similarity measure and the LSH technique. These new mechanisms should help to better represent the similarities between functions, preventing many failed merges, all the while boosting the time spent ranking functions in real-world programs. In the next sections, we show how we transform LLVM IR instructions into integer values, efficiently perform MinHash, create the hashmap, adaptively define a threshold per program, and redefine how phi-nodes affect the profitability of functions.

An overview of our proposed search strategy is shown in Figure 3.3.

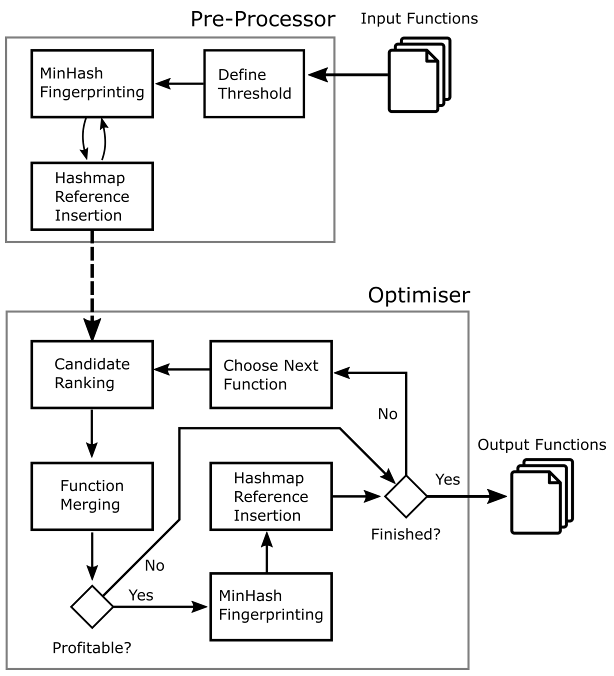


Figure 3.3: Optimisation Overview

3.3 Instruction to Integer

In the context of function merging, the crux of the MinHash algorithm rests on our ability to transform an LLVM IR instruction into an integer such that two matching instructions will have equivalent integer values and mismatched instructions will have differing values. In chapter 2, the generalised rules for matching two instructions had been listed: they must be equivalent in their opcodes, types, number of operands, and pairwise operand types.

Both opcodes and the number of operands are provided by LLVM as positive integer values. This makes representing these attributes very simple. However, types are more complicated. LLVM provides a TypeID system that defines all the base types for the type system. Fortunately, TypeIDs are just unsigned enums, which can be converted through a standard cast. But TypeIDs are often not accurate enough, with many instructions matching their TypeIDs but actually mismatching their true type. Instead, we combine the TypeID with the pointer value towards the memory location which contains the exact type. Since each individual type is stored once in only one memory location, we can guarantee that two matching types will have matching pointer addresses.

Pairwise operand types are handled in an identical manner. However, because the ordering of the operands matter, we make sure to distinguish them by multiplying their value by their operand position. Just these few transformations result in a decent representation of an instruction. Through extensive testing, we have found that this results in no false negatives (falsely mismatches two instructions), only false positives (falsely matches two instructions). The transformations above are lacking in depth to accurately capture the details in specific instructions that might make them unequal.

We then investigated increasing the depth that this transformation can go. Utilising most of the approaches previously discussed, we encoded more and more properties that determine the equality between these instructions. Through similar testing, we found far fewer false positives and a negligible number of false negatives.

This should suggest that this representation will better represent the MinHash similarities between functions. However, we have found that as we become stricter, the similarities between many profitable functions drops to near-zero. Their very low similarity scores make them very hard to discern between the dissimilar ones. We are not entirely sure as to why this happens but we think that even if two instructions do not match exactly but match with their generalised rules, they hold some amount of information about the rest of the function which indicates that the other instructions are much more likely to match.

While the increased depth should serve to reduce the number of failed merges more significantly than just applying the generalised rules, we have found that they do almost nothing to help improve the performance of the system. In chapter 5, we show that increasing code reduction can actually speed up the compilation process. This allows us to use the generalised rules only, giving us superior code reduction capabilities while not having to discount on the performance of the system.

Lastly, some instructions must be ignored during this process. As briefly explained in chapter 2, SalSSA does not merge phi-nodes, choosing to dismiss them from the sequence alignment equation and re-integrate them when the final merged function is being built. Therefore, phi-nodes should play no role in the similarity between two functions. Hence, we simply ignore them, pretending they were not a part of the function.

3.4 MinHash

In chapter 2, the MinHash algorithm was briefly described as an estimator for the Jaccard Index and how it can compute similarities between sequences in constant time. This is only one of various reasons why MinHash suits a searching similarity scheme.

MinHash utilises set similarities based on subsequences, which means it takes into account a fraction of structural similarity. The elements of each set are fixed-sized runs of instructions. Two elements are only equal if and only if all the individual instructions are equivalent and in order. With subsequences of instructions, the context for which they belong is better represented, and true similarities can be uncovered.

Additionally, the SalSSA system conceptually fails to solve the issue of storage. LLVM IR only has about 64 different opcodes making the storage of their frequency constant per function. However, because types are represented as just the set of involved types, if a function implements a new type on each instruction, then the storage of the set is bounded by the size of the function. On the other hand, MinHash reduces a function to a constant k hashes regardless of the original function size, resulting in constant storage per function.

3.4.1 Efficient Implementation of MinHash

To effectively utilise MinHash as a similarity measure, it is critical that we can efficiently compute the necessary hashes and similarity scores at incredible speeds. In this short subsection, we detail the most important optimisations made to the MinHash algorithm to rapidly boost its processing time to realise the full capabilities of the algorithm. We also provide one extra optimisation in the form of an early exit in the similarity computation. This optimisation can be found in appendix .1.

3.4.1.1 MinHash Variant

As discussed earlier, there are two variants of the MinHash algorithm which can estimate the Jaccard Index. Both versions were experimented with, and while the single-hash variant proves to be the fastest version, it cannot be applied.

The single-hash algorithm relies on at least k elements within the set to draw its hashes from. If there are $< k$ elements within the function, then it cannot produce the full k hashes that we desire. Function sizes can range from just 1 instruction to tens of thousands. If a function does not contain at least k elements then it will not be able to generate k hashes.

We experimented with ideas that involved artificially lengthening a function by repeatedly concatenating the function onto itself until it had enough elements. But this seemed only to work momentarily, deeming far too many dissimilar functions as similar. So, we proceeded with the multiple-hash variant but took special care to optimise it as best we could.

3.4.1.2 k Hash Functions

The first step in implementing the multiple-hash variant is choosing hash functions that are both fast and complex enough such that they are robust to permutations, i.e. the hash values depend on the ordering of its constituent characters. It may seem absurd to have k *good* hash functions, especially considering development time and the computational time of sufficiently complex hash functions. But at this stage, we can cut corners significantly and reduce the number of *good* hash functions to just 1.

If we hash each element with a *good* hash function and save the hash of each element as a base state, then we can utilise simple XOR operations with randomly generated numbers to compute our hashes. This will maintain the quality of our hashes while being much less expensive. Other operations like bitwise rotations (or shuffling) to generate the hashes are insufficient. Bitwise rotations will likely result in choosing the same element over and over again. Combining XOR with bitwise rotations could result in better hashes but the extra CPU cycles for little improvement in hash quality makes it unnecessary.

The chosen base hashing algorithm was FNV-1a as it is well known, widely used in practice, and also inexpensive. See code for FNV-1a in listing 3.1.

```
uint32_t fnv1a(const std::vector<uint32_t> &Seq)
{
    uint32_t hash = 2166136261;
    int len = Seq.size();
    for (int i = 0; i < len; i++)
    {
        hash ^= Seq[i];
        hash *= 1099511628211;
    }
    return hash;
}
```

Listing 3.1: FNV-1a Algorithm

In one iteration, the MinHash algorithm computes the FNV-1a hash of each element and temporarily saves them. The minimum hash is also saved in this operation so we only have to apply $k-1$ random numbers. The next $k-1$ iterations take these base hashes and XORs with pre-determined randomly generated numbers, saving the smallest hash on each iteration. This generates the k random hashes with only one inexpensive hash function.

3.4.1.3 Pipelining

A forgotten, and subsequently slow, step in the MinHash process is generating the shingles (subsequences) and their base hashes. A naive implementation would loop through each character in the sequence, computing the shingle where that character defines the starting point and then computing its hash. This algorithm revisits the same characters multiple times for each successive shingle, wasting many CPU cycles.

What should be noticeable is that generating shingles should only require visiting each character once, i.e. a streaming algorithm. We can accomplish this by storing partially computed hashes. On a read of a new character, the partially computed hashes are updated and the top hash is collected as the shingle's finalised hash. This idea can be seen in Figure 3.4.

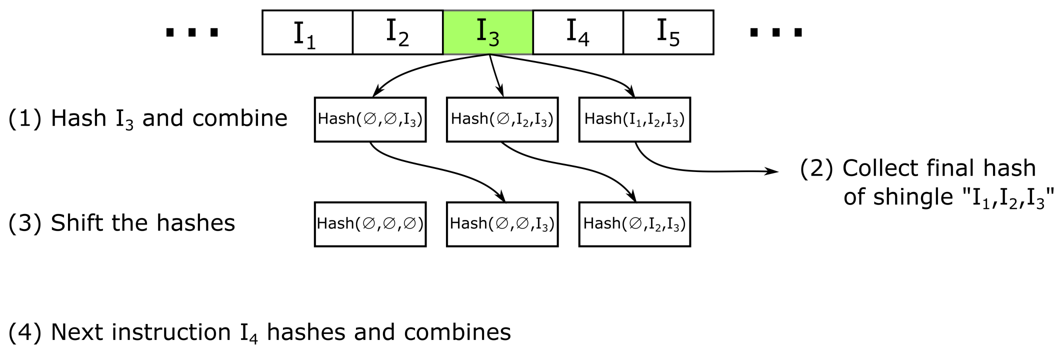


Figure 3.4: Efficiently Generating Subsequence Hashes

In Figure 3.4, the algorithm first hashes instruction 3, I3, and saves it in the first partially computed hash, and combines it with the second and third partial hashes. As the shingle beginning with I1 has been completed, it can be collected as its finalised hash value. The boxes containing the partially computed hashes are shifted and the process repeats itself.

On a test-bench of over 500,000 strings, each with 1,000 characters, this algorithm can save $\sim$15% on the pre-processing stage, compared to a standard implementation. The main speedup doesn't purely come from the prevention of revisiting characters, instead primarily coming from the compiler unrolling the loop which combines the hash with each partially computed hash. The instructions of the unrolled loop can be executed in parallel as they are independent of one another, giving far greater performance than other implementations we experimented with.

3.4.1.4 Choosing k and Memory Usage

The speed of both the pre-computation and pairwise calculations are dependent on the number of hashes that we use, k. But so does the accuracy of the estimation and the memory required to store every hash. We need to choose a reasonably sized value for k such that computations involving MinHash are fast without compromising on accuracy, and without needlessly using large chunks of memory.

The expected estimation error is bounded by $\mathcal{O}(\frac{1}{\sqrt{k}})$. At 400 hashes we can expect a maximum error of 0.05 in our estimation of the Jaccard Index. But 400 hashes require quadruple the computation time against just 100 hashes which only doubles the expected estimation error to 0.1. What's more, each hash is stored as a 32-bit unsigned integer. At 400 hashes, each fingerprint would require 1600 bytes, around 1.5KB per function. If we assumed a real-world stress-test scenario where the system is compiling a program with around 500,000 functions, storing just the fingerprints would take over 750MB. However, if we used only 100 hashes, this is significantly cut to just 190MB.

Ultimately, we chose $k = 200$ as we felt anything below 200 did not provide a dependable estimation, and anything over 200 quickly sees little to no estimation improvements and far worse computational and memory requirements. 200 hashes guarantee a maximum error on the estimation of 0.07, and each fingerprint uses only 800 bytes. We evaluate this decision in chapter 4 on the achieved code reduction and the consistency of those results.

3.5 Locality Sensitive Hashing and Searching

Now that we have an efficient similarity measurer between two functions, we can attempt to use the LSH technique to implement a search strategy that is both accurate and fast.

3.5.1 Efficient LSH and Searching

Likewise to the MinHash similarity scheme, the LSH search strategy must be optimised as much as possible to minimise the time it takes to compute the hashmap and find candidate pairs. In one massive pre-computational stage, every function fingerprint and band hash must be computed, storing its reference in a hashmap, and allow for rapid access speeds for similarity searching. Moreover, the hashmap must be able to quickly insert newly merged functions and delete fully processed ones.

3.5.1.1 HashMap and Buckets

LSH's reliance on a hashmap that allows for quick insertion, deletion, and access, makes conventional hashmaps undesirable. As we compute each function fingerprint, we store a reference to that particular function in the locations defined by its band hashes. This is ordinarily undertaken in the pre-computation stage where all input functions are processed and stored, but also when we add newly merged functions to the hashmap. Furthermore, we must be capable of removing functions from the hashmap when they have been fully processed.

Due to the intense demand on the hash map, we should choose to avoid C++'s STL unordered_map in favour of other implementations which greatly improve the performance in those aspects. Based on an article comparing the performance of many publicly available hashmaps [30], we decided to proceed with the Tessil robin_map [31]. It touts superior speed efficiencies in almost all cases - insertions, deletions, accesses - at the cost of slightly increased memory, however, this cost doesn't come into play until programs reach exceptionally numerous function counts.

This hashmap is open-source, license-free, interfaced after STL's unordered_map, and provided in a simple include folder, making it very easy to implement. However, this map is built only to store one item per bucket, as with many hashmaps. We have to simulate the bucket with a data structure with efficient iteration, insertion, and deletion speeds. This issue is also tightly connected to the problem of storing duplicate references to functions, where two distinct bands have the same hash value. A function with two or more equivalent bands will store multiple references to itself in that bucket, repeating the number of times the search strategy checks against this one particular function.

We opted to utilise the STL vector for its superior performance in iteration thanks to how it stores elements in contiguous memory. This has proven to be the deciding factor in the performance of the search strategy. Additionally, the vector is very attractive with its amortized constant time insertion and far reduced memory requirements compared to other alternatives. Furthermore, we alleviate the issue of duplication by filtering the bands of a function, thereby removing any duplicates and preventing the search strategy from checking the same buckets multiple times. We can do this efficiently since the number of bands is likely going to be small, maxing out at $k = 200$ but usually far smaller than this. The brunt of any apparent slowdown this causes is during the pre-computation stage, unlike the consistent searching slowdown caused by other alternative data structures.

We also limit the total number of elements that one bucket can hold to 100. This prevents code-bases, with exceptionally large numbers of similar functions, flooding the same buckets repeatedly. When this happens, the search strategy has to linearly scan through each one for each band. In effect, nullifying any potential performance gains that LSH is bringing. This constraint has shown to help performance in many benchmarks while proving to show little to no negative side effects on code size or the number of successful merges. However, we only apply this limitation to the initial pre-computation stage. We allow any newly merged functions to be added to the bucket even if the bucket exceeds 100 elements. This constraint also helps to solve the issue of deletion when using a vector, since deletion is linear to the size of the vector, as now the performance impact is limited.

3.5.1.2 Adaptive Threshold

Possibly the most egregious performance mistake the FMSA and SalSSA systems make is that they do not implement a threshold, attempting to merge each function with another, no matter how dissimilar they may be. Though, optimally choosing a threshold that maximises the number of successful merges and minimises the number of failed merges is a frustratingly difficult task. One shared threshold can be detrimental to the performance in some applications while beneficial to others.

Instead, our adaptive threshold bases itself on how willing we are to accept failed merges, and their associated time cost, in an attempt to elevate the ratio of successful merges to the code-base size as much as possible. In a program of only 100 functions, we may be willing to try nearly all merge attempts because the performance hit will be limited, suggesting a low threshold would be suitable. However, in a program with over 100,000 functions, function merging cannot afford to consider every single possible merge, choosing to consider only those that are guaranteed to be successful.

With this in mind, our threshold, t, defines itself on an expression based on the number of functions within the code-base, Eq 3.3.

$$t = \frac{1}{2+(\frac{x}{1-x})^{-3}} + 0.125 \tag{3.3}$$

where $x = \frac{log_{10}(nFunctions)}{10}$, and nFunctions represents the number of functions within the code-base. The hyper-parameters of this expression were set to allow the threshold to exhibit a maximum difference of around 0.5. This was chosen from testing involving the smallest and largest sized benchmarks, and finding thresholds that suited these programs well. This expression caps out at just over 0.6 and was chosen to allow even the largest of benchmarks to realise significant code reduction.

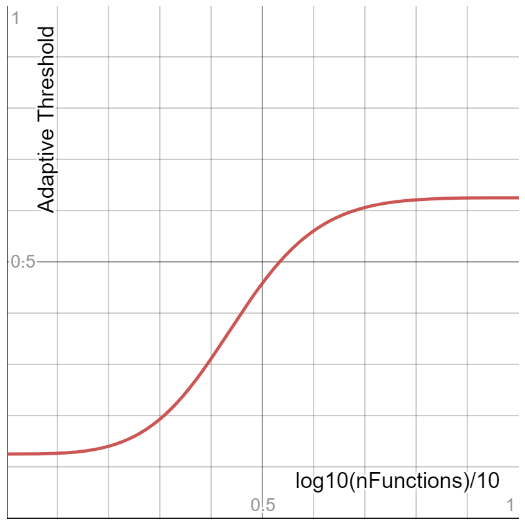
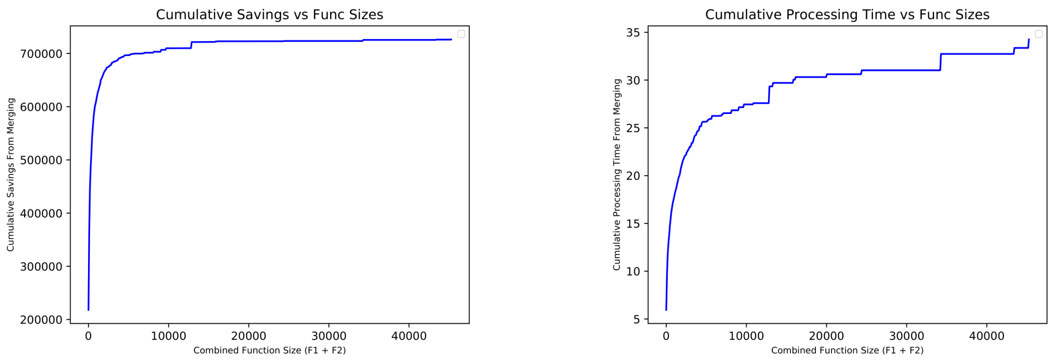


Figure 3.5: Adaptive Threshold Initialisation

This function follows an S-curve as can be seen in Figure 3.5. We opted for a sigmoid curve, instead of any other function e.g. linear, as an expression that exhibits small thresholds that smoothly accelerate in value proved to obtain the most promising results. We found through limited testing that a linear function too sharply rose the thresholds in smaller sized code-bases.

Consequently, these programs suffered in code reduction. The smooth sigmoid function allows us to keep the thresholds low for the modestly sized programs and rapidly increase for the larger ones.

An extra constraining threshold we can place on the code-base is on the processing of excessively large functions. Investigations have shown that as combined function sizes increase, $F1+F2$, the savings by merging these two functions tend to increase too. However, further investigations have shown that the merging of exceptionally large functions suffers disproportionate cumulative processing times when considering their code reduction capabilities. See (a) and (b) of Figure 3.6.



(a) Cumulative Merge Savings (b) Cumulative Merge Processing Times

Figure 3.6: Disproportionate Processing Times for Very Large Functions

Most code reduction is found in the merging of two functions with combined sizes ranging between 2 and 10,000 units because there are just so many of them. Yet, the cumulative processing times do not reflect this. The immense processing times of these exceptionally large functions do not warrant the minuscule amount of code reduction that they provide.

Therefore, we disallow any function to be merged with another such that their combined size is greater than 10,000. This also affords us the opportunity to completely ignore any function over 10,000, removing these functions from consideration. This has the added benefit of speeding up the pre-processing stage as these offending functions dominated the total time spent in that stage.

3.5.1.3 Choosing b and r

In chapter 2, the probabilities for which candidate pairs could be found through LSH was discussed. We saw that varying the values of b and r affected the probability that a data-point would be able to find another specific data-point, where $(\frac{1}{b})^{\frac{1}{r}}$ approximated the point at which they are 50% likely to find each other.

Given our previous discussion on the adaptive threshold, we must aim to pick appropriate values for b and r such that function similarities above our adaptive threshold are very likely to be found. However, we are restricted to just a select few choices.

Recalling that b and r are bound by the number of hashes in the fingerprint, b and r must multiply to give k. For example, at $k = 200$, the lowest LSH threshold we can assume is given by $b = 200$, $r = 1$, resulting in $t = 0.05$.

Also noting that our adaptive threshold maxes at 0.6125, this affords us only the b and r combinations found in Table 3.1. Decreasing the band size further results in LSH thresholds above 0.6125, making them unusable.

The bands and row sizes maintain an intimate relationship with how performant our searching strategy can be. Increasing the number of bands dictates increased accesses into the hashmap resulting in many more comparisons. This is where an unfortunate aspect of our adaptive threshold can be found, smaller sized code-bases are given smaller thresholds resulting in increased hashmap accesses and increased searching times.

b	r	$(\frac{1}{b})^{\frac{1}{r}}$
200	1	0.05
100	2	0.1
50	4	0.376
40	5	0.478

Table 3.1: Potential b and r Combinations

To mitigate this, we only apply LSH if the total function count exceeds 100. This way, we get the iteration speed of the linear scan when function counts are sufficiently small that using LSH would only be detrimental to performance, and get the drastic performance boost when linear scans are inefficient. In these select cases, it affords us the removal of the hashmap pre-processing and storage. However, we maintain the use of MinHash as the similarity measure, and the fingerprints must still be computed.

As for function counts over 100, we decided to define b and r based on these possible ranging values of the adaptive threshold, t:

$$\begin{aligned} b &= 200, r = 1 & t &< 0.15 \\ b &= 100, r = 2 & 0.15 \leq t &< 0.4 \\ b &= 50, r = 4 & 0.4 \leq t &< 0.5 \\ b &= 40, r = 5 & 0.5 &\leq t \end{aligned}$$

The first combination is rarely utilised. More often than not, the linear scan is employed as function counts typically don't reach 100 in that threshold range. Discounting that combination, the probabilities for which candidates could be found at the minimum similarity in their specific threshold range are ~90%, ~73%, and ~72%. The ranges have not been optimised for guaranteed candidate pair lookup but were selected because these probabilities are *good enough*.

3.5.1.4 Thunk Predictor

Computing the effective size of a merged function is not purely determined by the number of instructions within that function. In a perfect world, when two functions are merged, they can be deleted and all calls to those functions are replaced with a call to the newly merged function. However, it is often the case that we cannot delete the original functions nor replace all the calls to those functions. This can be caused by the program storing references to one of these functions. Deleting one could break the program.

In these cases, the body of each offending function is replaced with a single call to the merged function, maintaining the original calls to those functions. When the program calls for one of the component functions, it perceives no difference. But once inside the body of that function, it calls the merged function. This extra jump or extra call is known as a thunk. The instructions in a thunk have an associated code size cost to them. That is why, when computing the profitability of merging two functions, it is important to check if the thunk has such a negative impact on code size that it actually increases the total code size.

The observation we can make here is that when taking into account a thunk's cost, even a 100% match on two functions may not be enough to overcome that cost. Since a thunk's cost is independent of the merged function, we can find it before performing the expensive sequence alignment and merging operations. If two functions satisfy this relationship then they will never be profitable to merge and can be aborted early.

$$max(F1,F2) + ThunkSize(F1,F2) > F1 + F2 \tag{3.4}$$

Unfortunately, this check will only result in a small boost in performance, since the functions which could satisfy this relationship are going to be very small and quick to operate on. But as code-bases grow, every small speedup will begin to add up significantly. This will also drastically improve the accuracy of our search strategy, boosting the ratio of successful merges to failed merges.

3.5.2 Memory Usage

The performance of the LSH technique comes at the cost of memory. We mentioned that each fingerprint stores 200 32-bit hashes, totalling around 800 bytes of memory per function. But this is forgetting the storage of the band hashes. We do not want to have to re-compute a function's bands every time we want to search with it. Band hashes are, at max, equivalent to the storage of each fingerprint. This happens when the row size is set to just one hash and the number of bands is k. Now, storing a function fingerprint and its hashes is bounded by $\mathcal{O}(8k)$ bytes, or in our case, 1600 bytes.

But it's in the hashmap where we compromise memory for the speedups that LSH provides. Assuming no memory overhead to using the Tessil robin_map and no memory usage from empty buckets, the memory requirement of the hashmap is bounded by $\mathcal{O}(32Lb)$, where b is the number of bands and L is the number of functions being stored. We suppose this worst-case conclusion by assuming no function is similar to another, allocating just one element per bucket. A vector utilises 24 bytes in a 64-bit system and storing one function pointer is 8 bytes. Re-using the theoretical real-world stress-test scenario of 500,000 functions, the hashmap could, at max, require just under 3GB of memory. We are likely to use more, however, with the storage required by the hashmap implementation itself.

While this may seem quite horrendous, it's not otherworldly for compiling a program whose size rivals that of some of the largest pieces of software ever made. Even so, this estimate is a worst-case estimate. As code-bases get larger, the number of bands can be tuned down to allow for a stricter threshold in the LSH system, reducing the memory usage. For instance, a reasonable row size at 500,000 functions would be around 5, giving a band size of just 40. The memory usage would then become just 610MB.

3.6 Cost Profitability Fine-Tuning

The final stage in merging two functions involves checking that integrating the merged function will give a lower total code size. This is simply done by adding up the instruction costs within F1 and F2 and comparing against the cost of the merged function, M.

One LLVM IR instruction is not guaranteed to be translated into one machine instruction, and machine instruction costs are not equal to one another. To account for this, FMSA and SalSSA use LLVM's provided target-transformation interface (TTI) to estimate the cost of each instruction, and subsequently the whole function. However, due to the major overhaul in the way SalSSA handles phi-nodes, it weighs these specific instructions differently.

Instruction costs are given without units and do not directly translate to binary sizes. However, comparing costs will adequately indicate whether one is likely to result in an increased or decreased binary size when compared to another. With that said, in SalSSA, phi-nodes are given a cost of 0.2. When dealing with fractional instruction costs, SalSSA computes the aggregate cost of a function and then takes the ceiling value, which then gives the expected integer cost.

During the development of the search strategy, we found that, when merging two functions, the number of phi-nodes within the merged function is massively increased. Considering the SPEC2006 benchmark suite, the number of phi-nodes within F1 and F2 combined is increased nearly nine-fold in the merged function. This is due to how SalSSA generates additional phi-nodes to retain the SSA form. Consider Figure 3.7, where two functions have matched on the last half of their singular basic block, resulting in the merged function as shown in the figure.

The differing first halves must be gapped and result in the branching flow of execution as seen. Unfortunately as a result, SSA form may become violated. The shared instructions and their variables may have separate definitions from either incoming branch, or variables may be defined in only one of the incoming branches.

SalSSA solves both of these issues by inserting a pseudo-definition in the entry block, guaranteeing that all definitions of a variable are dominant. It later simplifies these definitions through a novel process called phi-coalescing [4]. But these conflicting definitions must be resolved. In the uniting basic block, every violating variable must be resolved with phi-nodes. If the number of defined variables in the gapped blocks becomes large, then so does the number of generated phi-nodes.

This simplistic case does not represent the sheer potential for when phi-nodes must be generated. As the complexity of the merged function grows, the number of generated phi-nodes also grows. More branches in control flow results in more phi-nodes being generated. But recalling the discussion on phi-nodes in chapter 2: additional phi-nodes should not present an issue for code size as they are not directly translated to binary code in later compilation stages.

The problem is, the current cost of phi-nodes is given a fixed size value of 0.2. So, when sufficiently complex functions are merged, the merged function is flooded with phi-nodes and deemed not-profitable because it contains so many of them. A simple fix would involve setting their cost to 0, allowing some applications to deem many more functions as profitable. But we shouldn't forget that when they become seriously numerous, they do end up having a code size cost with their increase in load and store instructions caused by *register spilling*. Load and store instructions usually have a higher code size cost to them than instructions that access the registers.

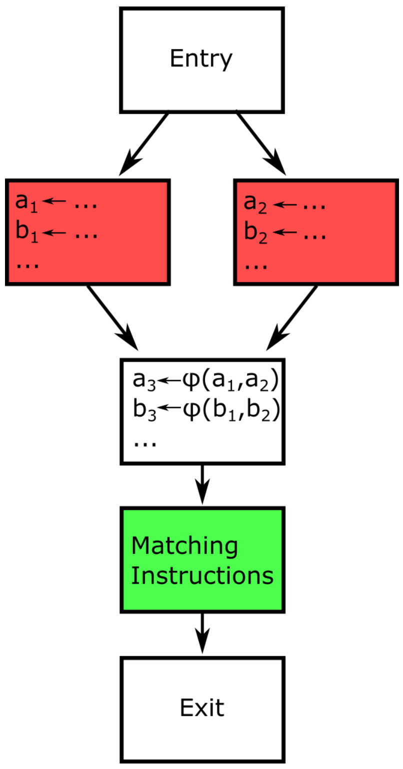


Figure 3.7: Resolving New Branching Flow With Phi-nodes

We then count phi-nodes by ramping up their impact as they increase in number. Per phi-node in a function, we weigh its cost by how many have come before it. See code listing 3.2 to see how we estimate the size of a function with this new weighting. The weighting 0.0002 was chosen because it produced reasonably good results but has not been experimentally proven to be the best weighting.

```
size_t EstimateFunctionSize(Function *F, TargetTransformInfo *TTI) {
  double size = 0;
  int count = 0;
  for (Instruction &I : instructions(F)) {
    if (I.getOpcode() == Instruction::PHI){
      count++;
      size+=(count*0.0002);
    }
    else{
      size += TTI->getInstructionCost(
          &I, TargetTransformInfo::TargetCostKind::TCK_CodeSize);
    }
  }
  return size_t(std::ceil(size));
}
```

Listing 3.2: Function Size Estimation

Chapter 4

Evaluation

In this section, we evaluate the proposed search strategy and analyse our improvements on code reduction, compilation time, and memory usage. The proposed optimisation is presented alongside the current state-of-the-art, SalSSA [4], and a baseline that applies no function merging optimisations. We also present statistics relevant to the function merging procedure. Code reduction is presented as the decrease in code size compared to the baseline system. Similarly, compilation time is presented as normalised to the baseline compilation time, which allows for direct comparisons. Memory usage represents the peak maximum memory usage seen by the compilation process. Code size reduction and memory usage are presented with a geometric mean whereas compilation time is presented with an arithmetic mean.

We evaluate the performance of these systems through the C/C++ SPEC 2006 benchmark suite [1], and a couple of select large-case programs including the LLVM codebase and the Linux kernel v5.11. The LLVM program is compiled without the frontend procedure of the compiler, and so compilation times are presented as the combined time of the optimiser and the code generator. On the other hand, the Linux program is only presented with the compilation time of the optimiser. There is an unknown bug that prevents the backend from successfully compiling in both SalSSA and our method. All experiments have been performed on a dedicated server, intended to minimise noise, with specs:

- Cpu: Intel Xeon CPU E5-2560 v2 @2.6GHz, 16 cores
- Memory: 64GB
- OS: Linux ”Ubuntu” v18.04.3 (Bionic Beaver), kernel 4.15.0-69-generic

The function merging optimisation runs as a pass within llvm, version 11.0.0, with clang, version 6.0.0-1ubuntu2. Due to the randomness involved in the MinHash and LSH algorithms, we present the average results of 10 distinct runs. The error bars represent the variability of the results given by the standard deviation that is centred around the mean of those results.

We chose to optimise for code reduction by only applying the generalised rules of the IR to integer transformation, and setting $w = 2$. The shorter the subsequence size, the more likely it is for two subsequences to be equal to one another. These parameters are selected to allow the system to find as many successful merges as it can while keeping compilation times as low as possible.

Figure 4.1 shows the code size reduction over the baseline. Both vanilla SalSSA [4] and our proposed optimisation achieve similar code size reductions. Our improvements observe an increase of code reduction by 0.4% in the geometric average. We have found that both SalSSA and our method frequently choose the same successful pairs of functions for merging. The difference in code reduction comes from the edit to cost profitability with how we weight phi-nodes, allowing more merges to be deemed profitable and deliver minor boosts in code reduction.

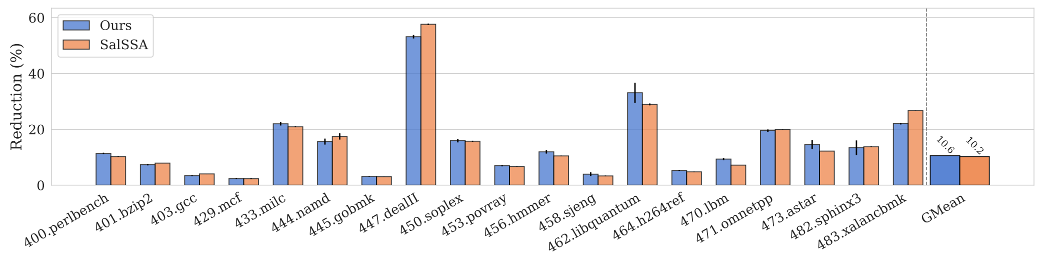


Figure 4.1: Code Reduction

But where we see less code reduction is mostly found in larger benchmarks like DealII and Xalancbmk, containing 7380 and 14191 functions respectively. With these bigger benchmarks, the adaptive threshold sets itself relatively high, causing them to miss out on some profitable functions. Decreasing the threshold gives back the remaining code reduction but, by that point, we start to feel the negative effects of failed merges and their processing times.

Figure 4.2 shows the compilation time normalised to the baseline. We see significant performance improvements over the state-of-the-art and speedups compared to our baseline. In most cases, our optimisation completely nullifies the overhead brought about by function merging. Unexpectedly, we achieve compilation times faster than the baseline compiler performing no function merging at all, averaging 4% faster than the baseline.

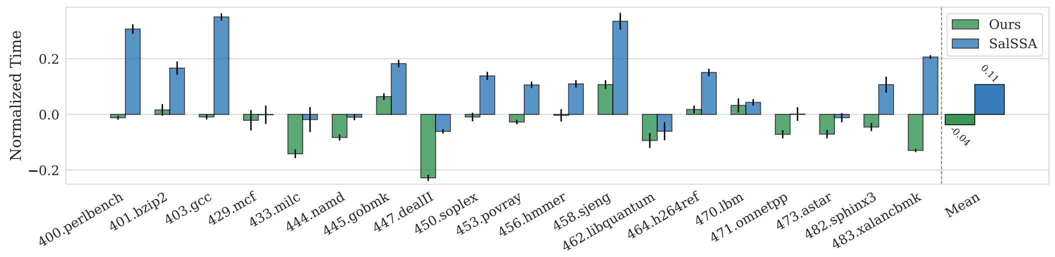


Figure 4.2: Normalised Compilation Time

Looking at the DealII, and Xalancbmk benchmarks, the reason for the notable improvements over SalSSA comes principally from the LSH lookup scheme. These benchmarks gain a lot of speedup from using the LSH hashmap to find similar functions because of their size. However, many other benchmarks report remarkable improvements where LSH isn't the sole reason. Most of the smaller sized benchmarks, and including the large ones, enjoy the reduced number of failed merges thanks to the adaptive threshold and MinHash similarity measure.

Analysing the benchmarks which are still slower relative to the baseline, like gobmk, sjeng, and lbm, reveal a couple of problems. Both sjeng and lbm still suffer from the majority of their processing times being dedicated to failed merges. These processing times can be found in Figure 4.3. On the other hand, the gobmk benchmark is slowed down by how many *similar* functions it contains. Hundreds of functions are deemed similar by LSH and placed in the same buckets, causing massive lookup times. Even the cap on bucket sizes at 100 functions does not fully manage to mitigate this issue. Unfortunately, capping the buckets any lower than this starts to display negative effects on code reduction.

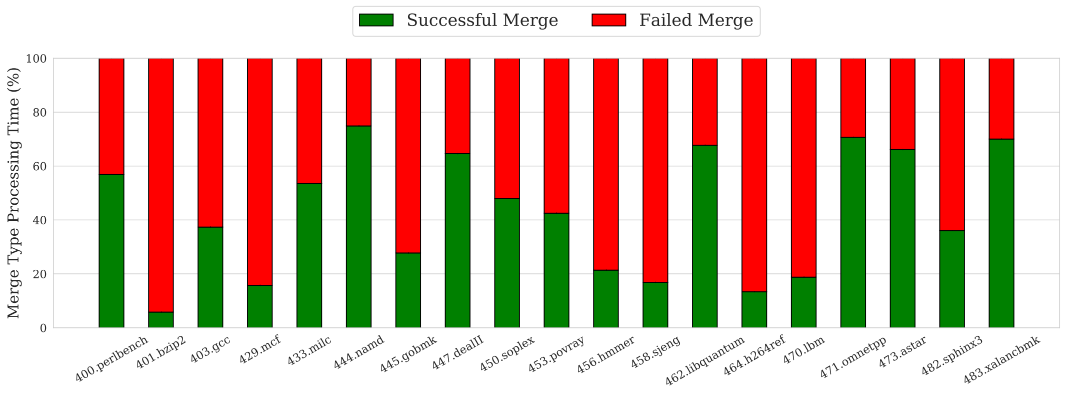


Figure 4.3: Processing Times of Successful vs Failed Merges

Comparing the performance of both systems reveals how much of an improvement has been made. Figures 4.4 and 4.5 shows how our optimisation manages to achieve increased average successful merges while decreasing the number of failed merges by ∼4.3x. Figure 4.6 also shows how much of an improvement has been made on the total processing time of failed merges. Less than a fifth of the time is spent on failed merges than in SalSSA. For reference, the average time spent on successful merges is exactly equivalent to SalSSA except in the cases where LSH has drastically reduced the ranking times in large benchmarks.

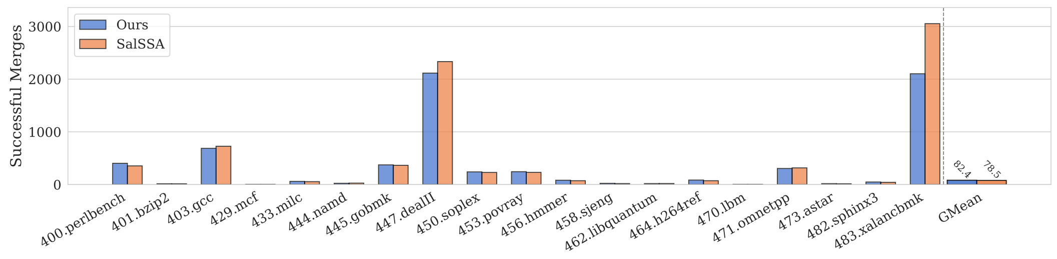


Figure 4.4: Number of Successful Merges

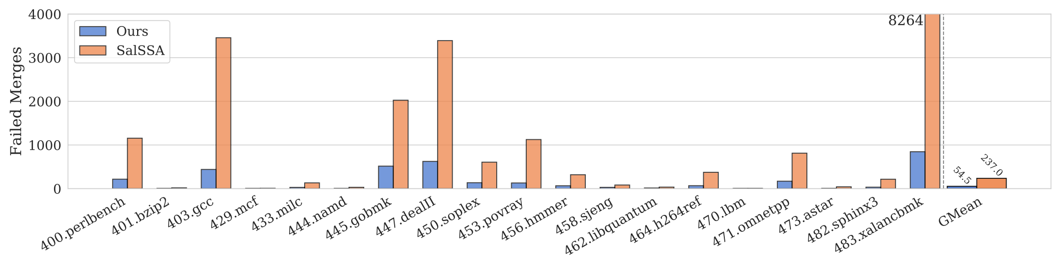


Figure 4.5: Number of Failed Merges

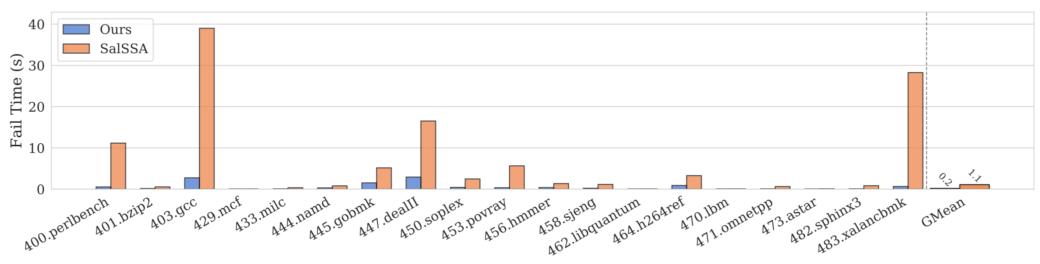


Figure 4.6: Time Spent on Failed Merges

What we deduce from these results is that failed merges endure disproportionately prolonged processing times compared to successful merges. This is why they typically still dominate the time spent in the function merging procedure despite how average successful merges outnumber average failed merges.

The alignment algorithm has free rein over how the merged function should look and it does not care about aligning contiguous instructions between many distinct basic blocks. By aligning instructions in this way, the resulting merged function will need to contain many branches to facilitate the new control flow. To do so, the code generation steps have to work very hard to produce a function that does not break the functionality of each original function, suffering lengthier times to complete than more straightforward alignments, as seen by the code generation and post-processing steps in Figures 3.2 and 4.7.

These merges are much more likely to fail due to exactly what makes them so difficult to build, with escalated code sizes caused by the increased number of branches and phi-nodes. In essence, failed merges dominate the overall time because the functions being produced are far more complex than their component functions, causing prolonged code generation times and reduced likelihood of profitability. We believe the reason for this is due to how the merging procedure can produce functions that are far more complex than their component functions, causing increased code generation times and reduced likelihood of profitability.

Figure 4.7 shows the breakdown of the function merging pass with our method. We see that the total time is still predominantly affected by the code generation and the post-processing steps. We see that fingerprinting has started to come through as a substantial step as we generate all the hashes and build the hashmap. The ranking

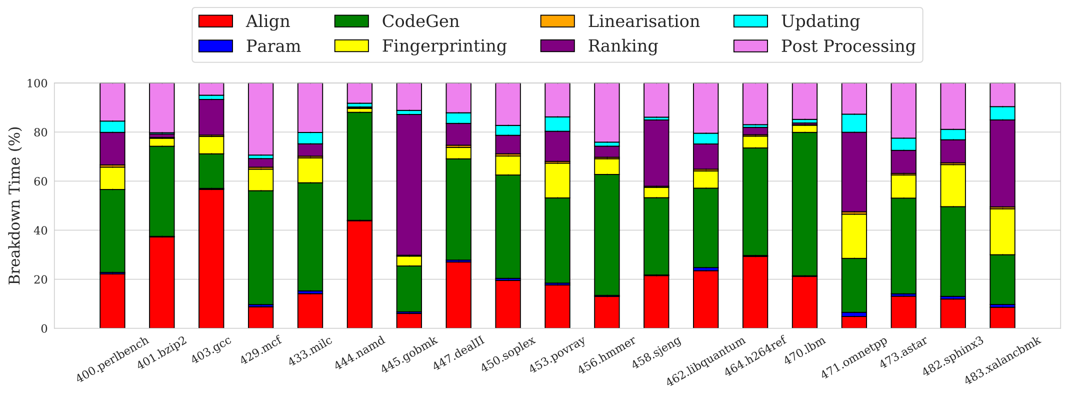


Figure 4.7: Proportion of Time Spent Doing X

k (Number of Hashes)	GMean Code Reduction (%)	Standard Deviation
50	10.3	0.3
100	10.4	0.14
200	10.4	0.08
300	10.4	0.08

Table 4.1: Varying k on Code Reduction

proportion seems in line with SalSSA, as in Figure 3.2, however, actually comparing them relatively shows that ranking is massively reduced. In fact, everything has been massively reduced thanks to the prevention of many failed merges.

An evaluation of our decision to use $k = 200$ on code reduction can be found in Table 4.1, averaging 3 distinct runs each. Due to the intricate relationship between k and the LSH searching scheme, we modified the system and only evaluate code reduction entirely on linear scans. It would be unfair to compare systems with un-optimised b and r values chosen for LSH. What we ascertain is that lower hash counts achieve just as good code reduction capabilities. However, as we include more hashes, the standard deviation of the resulting code reduction decreases, giving more consistent results.

While lower values for k may seem promising, with the benefit of reducing similarity computation and pre-processing times, they will impede our ability to effectively use LSH. Recalling the discussion in subsubsection 3.5.1.3, the number of available hashes dictate our ability to choose b and r for LSH. With lower hash counts, we are subjected to potentially worse LSH performance as the number of acceptable thresholds available to us are considerably reduced. This is conceivably the most compelling reason for maintaining $k = 200$: we get consistent code reduction and a decent selection of thresholds for our LSH hashmap. However, there may be a compelling reason to drop down to $k = 100$ and tweak b and r appropriately.

Figure 4.8 shows the peak memory usage during compilation time. Surprisingly, our improvements achieve lower average memory usages, but this is due to the obscene memory requirements seen in gcc. Investigating this benchmark shows that SalSSA attempts to merge two very large functions and the sequence alignment algorithm struggles to handle these functions efficiently due to its quadratic memory requirements. Our search strategy ignores these two functions due to their size and does not suffer from this. However, this could also be improved by applying Hirschberg's algorithm [14], reducing the memory requirement from quadratic to linear. Omitting this benchmark from consideration, we see that our proposed optimisation achieves more or less equivalent memory requirements to SalSSA, further demonstrating the lack of downsides to our method. It is intriguing to note how our optimisation achieves lower average memory usages over the baseline.

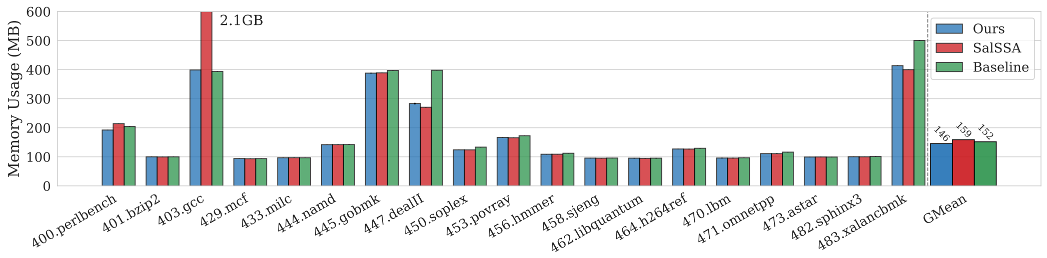


Figure 4.8: Peak Memory Usage During Compilation

Table 4.2 shows the results of applying our optimisation to the LLVM and Linux kernel code-bases. While all systems were run 10 times to achieve an average on each benchmark, SalSSA was run only once on the LLVM benchmark due to the extreme time it takes to fully compile. The sentiment remains the same.

LLVM (OPT + GEN)	Baseline	SalSSA	Ours
Code Reduction (%)	N/A	30.2	30
Compilation Time (s)	1011.48	11139.2s	859.24
Peak Memory Usage (GB)	9.86	8.34	7.3
Linux Kernel (OPT)			
Code Reduction (%)	N/A	N/A	N/A
Compilation Time (s)	25.97	1376.34	30.13
Peak Memory Usage (GB)	1.37	1.177	1.184

Table 4.2: Results on Large-Case Programs

In the LLVM benchmark, our method achieves an average compilation time speedup of $\sim$13x over the current state-of-the-art in addition to providing a boost over the baseline by $\sim$15%. Against SalSSA, our method comes with improved memory requirements, for the same reason as in gcc. These improvements come at the cost of just 0.2% in code reduction. Both SalSSA and our optimisation achieve lowered memory requirements against the baseline, lowering them quite significantly.

The Linux Kernel obtains similar results with a compilation time speedup of $\sim$45x over SalSSA but at the cost of $\sim$16% against the baseline. Again, SalSSA and our optimisation attain lowered memory requirements against the baseline, however, SalSSA outperforms our method by less than a per cent.

Chapter 5

Discussion

Our evaluation arises a burning question: how is it possible that we can achieve faster than baseline compilation times while also performing function merging? Recalling the LLVM compiler architecture discussed in chapter 2, the stages of the compiler work serially to deliver the finalised compiled binaries. Regardless of the optimisations performed in the optimiser, the frontend will always record the same timings. However, the same cannot be said for the code generator.

The code generator depends on the output of the optimiser. If function merging has managed to significantly reduce the volume of code, then the backend benefits from this reduced workload. Consider Figure 5.1, the relative time spent in the code generator phase is drastically reduced by both SalSSA and our method when compared to the baseline. Both achieve an average speedup of 12% over the baseline code generation stage. But SalSSA seldom benefits from this speedup due to the length of time spent within the optimiser performing function merging. See Figure 5.2 for the relative time spent within the optimiser.

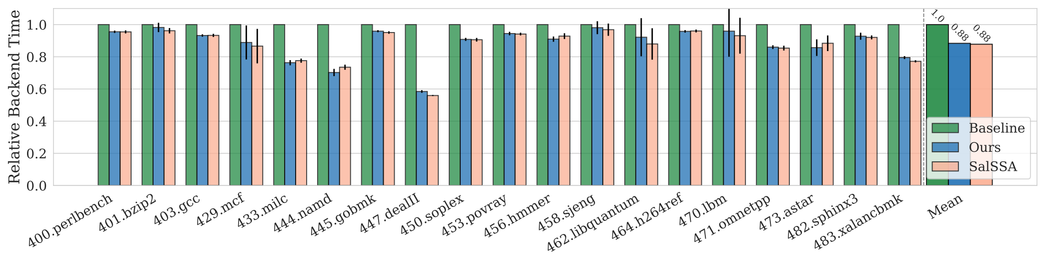


Figure 5.1: Speedup in the Backend

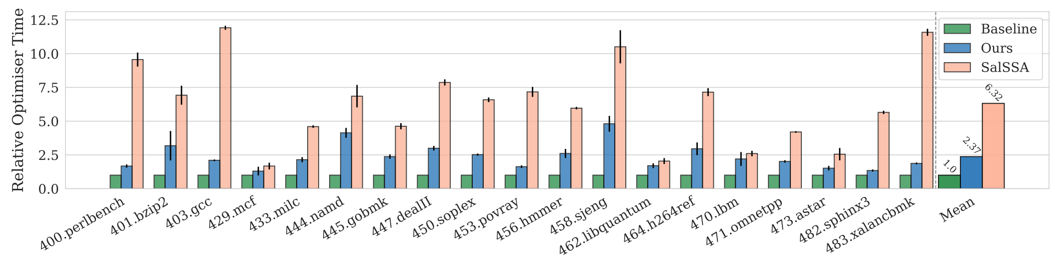


Figure 5.2: Optimiser Compilation Time Relative to the Baseline

Take for instance the LLVM benchmark, both SalSSA and our method achieve significant speedups in the code generator phase compared to the baseline, around 20% and 30% respectively, equating to under 4mins and 5mins. However, that speedup is completely nullified with SalSSA due to the near 3 hours it spends in the optimiser phase. Since our method only spends, on average, 2mins 20s in the optimiser phase, we benefit from the reduced workload issued to the code generator. Comparing purely the time spent in the sequence alignment function merging pass for LLVM, the speedup of our method over the current state-of-the-art exceeds well over 250x.[1]

There is a strong correlation between the bitcode size reduction, achieved by the function merging optimiser when compared to the baseline optimiser, and the relative speedup in the code generator phase. Figure 5.3 displays this correlation. As more savings are made through function merging, there is a decrease in time spent in the final stage of the compiler. So long as the function merging procedures can be performed swiftly, the whole compilation process will benefit from this reduced time.

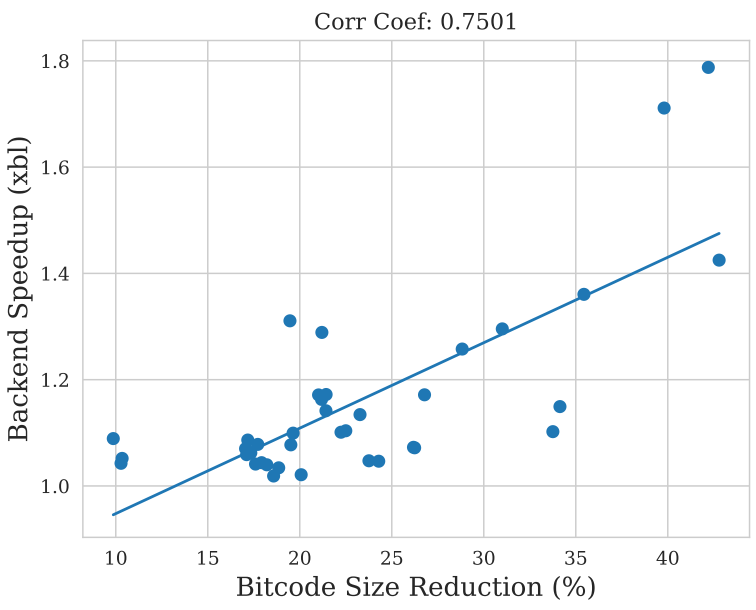


Figure 5.3: Bitcode Reduction vs Backend Speedup

[1]We average ∼35s for the function merging by sequence alignment pass whereas SalSSA sees ∼2hrs and 50mins.

Furthermore, the effect of the code generator on the total compilation time is far more significant than the optimiser. See Figure 5.4 for the proportion of time spent within each compiler phase with our method, after having significantly reduced the time spent within the optimiser compared to SalSSA. Knowing that the backend is responsible for much of the compilation time, it is actually beneficial to brunt slower optimisation times in an attempt to maximise code reduction so that the backend stage may be sped up as much as possible. In fact, we experimented with this a little and found that we could scarcely gain much performance, through raising the threshold etc., and promptly found deteriorating performance as the gains in the backend started to diminish. If we wished to show off speed in the optimisation phase, this would not be challenging to achieve, however, the performance of the entire compilation process would suffer as we lose the code reduction that boosts the backend.

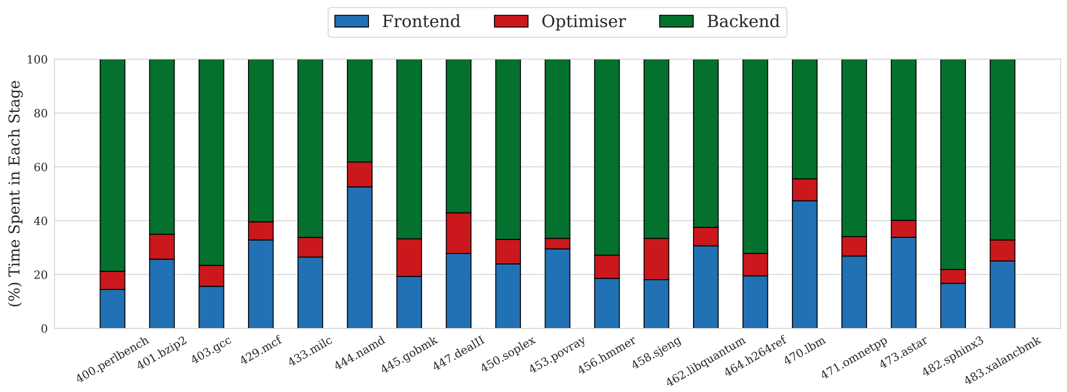


Figure 5.4: Proportion of Time Spent in Each Compiler Phase with Our Method

We equally observe similar results when it comes to the memory requirements of the whole compilation process. In many instances, the peak memory usage is dictated by the backend of the compiler. By reducing the total code volume in this phase, we see reduced peak memory usages compared to the baseline. Again using the LLVM benchmark as the most evident example, both SalSSA and our method achieve lower memory requirements, dropping the baseline peak size from near 10GB to 8.3GB and 7.3GB respectively, resulting in drops of around 15% and 25%. As with the gcc benchmark, our method lowers the required memory of SalSSA by ignoring the merging of exceptionally large functions and their associated quadratic memory cost.

It is for these reasons why the Linux benchmark doesn't achieve faster than baseline compilation times with our method. Because we don't apply the backend stage in compilation, we never enjoy the boosted speeds that allow total compilation time to overtake the baseline. Interestingly, SalSSA and our method still display better memory requirements over the baseline even though the backend plays no role. It is possible that there is a particular stage within the optimiser, which is performed after function merging, that benefits just as the backend would.

These extraordinary results allow us to present our method, not only as an optimiser for code size but also for improving both the performance of the compiler itself and the resources required to compile programs.

Chapter 6

Conclusion

In this work, our primary goal was to improve the performance of the current state-of-the-art in optimising compilation for code size. Through implementing a search strategy based on the MinHash and Locality Sensitive Hashing techniques, we have achieved vastly superior performance with notable improvements in code reduction and memory requirements. We have also shown that function merging can serve to reduce the workload of the backend stage in compilation, allowing for quicker overall compile-times and reduced memory requirements.

Applied to the SPEC2006 benchmark suite, we average faster than baseline results by 4%, with increased code reduction against the state-of-the-art, and improved memory requirements against both the baseline and SalSSA. Applied to real-world large-case programs including the LLVM code-base and the Linux kernel, we see dramatically improved performance and memory requirements, at little cost to code reduction. In effect, we have established practicality for all applications, eliminating almost all misgivings that currently impede widespread adoption.

6.1 Future Work

Considering Figure 4.3 from chapter 4, the next opportunity for further optimisation would be to minimise the disproportionate amount of time that is squandered on failed merges. However, as discussed in the same chapter, we believe that this is likely caused by the merging of functions into an extremely complex merged function, requiring exceptionally large code generation times, as seen in Figures 3.2 and 4.7. And as it currently stands, a recent paper has been submitted for review that combats this issue by reducing the alignment algorithm to the basic block level, allowing two complex functions to be merged without constructing a merged function that is increasingly complex. Currently, we are working with the authors of SalSSA, and this new paper, to combine our work in an upcoming research paper.

Bibliography

[1] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[2] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[3] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES'97*, pages 21–29. IEEE Computer Society, 1997.

[4] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Effective function merging in the ssa form. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 854–868, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22:378–415, 03 2000.

[6] Pinaki Chakraborty. Fifty years of peephole optimization. *Current Science*, 108(12):2186–2190, 2015.

[7] Chris G. Demetriou. Safe icf : Pointer safe and unwinding aware identical code folding in the gold linker. 2010.

[8] Ian Lance Taylor. A new elf linker.

[9] Tobias J. K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. Exploiting function similarity for code size reduction. In *LCTES '14*, 2014.

[10] Rodrigo Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function Merging by Sequence Alignment. In *Proceedings of the 2019 International Symposium on Code Generation and Optimization*, pages 149–163, United States, 2 2019. Institute of Electrical and Electronics Engineers (IEEE). Date of Acceptance: 29/10/2018.

[11] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[12] 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.

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

[14] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[15] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

[16] Eugene W. Myers and Webb Miller. Optimal alignments in linear space. *Bioinformatics*, 4(1):11–17, 03 1988.

[17] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[18] Robert D. Fleischmann Jeremy Peterson Owen White Steven L. Salzberg Arthur L. Delcher, Simon Kasif. *Alignment of whole genomes*, 14-03-2020. `http://mummer.sourceforge.net/MUMmer.pdf`.

[19] Chakraborty Angana and Bandyopadhyay Sanghamitra. FOGSAA: Fast Optimal Global Sequence Alignment Algorithm. *Scientific Reports*, 3(1):1746, 2013.

[20] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[21] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[22] Jaccard. The distribution of the flora of the alpine zone. In *New Phytologist*, volume 11, pages 37–50, 1912.

[23] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[24] Yiqiu Wang, Anshumali Shrivastava, and Junghee Ryu. FLASH: randomized algorithms accelerated over CPU-GPU for ultra-high dimensional similarity search. *CoRR*, abs/1709.01190, 2017.

[25] Anshumali Shrivastava and Ping Li. Densifying one permutation hashing via rotation for fast near neighbor search. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 557–565, Bejing, China, 22–24 Jun 2014. PMLR.

[26] Yury A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *CoRR*, abs/1603.09320, 2016.

[27] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations (extended abstract). In *IN STOC '98: PROCEEDINGS OF THE THIRTIETH ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING*, pages 327–336. ACM, 1998.

[28] Tyler Neylon. Introduction to locality-sensitive hashing. `http://tylerneylon.com/a/lsh1/`. Accessed: 2021-04-09.

[29] Jure Leskovec, Jeffrey D. Ullman, and Anand Rajaraman. *Mining of Massive Datasets*, page 73–134. Cambridge University Press, 2020.

[30] Tessil. Benchmark of major hash maps implementations. `https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html`. Accessed: 2021-04-09.

[31] Tessil. Tessil robin-map. `https://github.com/Tessil/robin-map`. Accessed: 2021-04-09.

.1 Quick Abort

The next place we can find optimisations in the MinHash process is the pairwise similarity computation. Given the two sorted lists of hashes, we simply need to iterate through them, only incrementing a set's iterator when a match occurs or their current hash is less than the opposite set's current hash. For interest, we couldn't find much optimisation opportunity in the sorting mechanism, opting to use the C++ standard library std::sort as it is fast, reliable, and requires no development time.

We can do better. The observation we can make here is that we don't want to carry on calculating the similarity between two functions when, according to the current stage they're at in the calculation, the maximum possible similarity is below a certain threshold. If we determine that the two functions are guaranteed a similarity below this threshold then we wouldn't want to bother with the rest of the calculation. This way, only similarities between functions larger than this threshold are fully calculated.

If we assume we have already processed pos_1 and pos_2 hashes in the first and second fingerprints respectively, len_1 and len_2 define the number of hashes in each fingerprint respectively, and the number of matching hashes is $nintersect$. Then the Jaccard Index estimate $J(s_1, s_2)$ is subject to this boundary:

$$max(J(s_1,s_2)) = \frac{nintersect + min(len_1 - pos_1, len_2 - pos_2)}{len_1 + len_2 - nintersect - min(len_1 - pos_1, len_2 - pos_2)} \tag{1}$$

Constraining our consideration of Jaccard estimations greater than or equal to some threshold, α, we can assert that any pairwise computation which satisfies this condition should be aborted, $max(J(s_1,s_2)) < \alpha$. We can rearrange this equation to:

$$nintersect + min(len_1 - pos_1, len_2 - pos_2) < \frac{\alpha}{1+\alpha}(len_1 + len_2) \tag{2}$$

If this condition is ever satisfied during the similarity calculation, then we can abort knowing that the two functions were never going to be similar enough to be worth considering. We can pre-compute the RHS of the condition and carry out the check on each loop of the algorithm.