WILL IT MERGE?

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2025

Tuan Chuong Goh

Supervised by Dr. Pavlos Petoumenos

Contents

Abstract 7

Declaration 8

Copyright 9

Acknowledgements 10

1 Introduction 11

1.1 Motivation 11
1.2 Aims 12
1.3 Report Structure 13

2 Background 14

2.1 The LLVM Compiler 14
2.1.1 LLVM Architecture 14
2.1.2 LLVM IR 15
2.1.3 LLVM Passes 15
2.2 Existing Function Merging Approaches 15
2.2.1 LLVM Function Merging 15
2.2.2 F3M: Fast Focused Function Merging 16
2.3 Machine Learning 21
2.3.1 Supervised Learning 21
2.3.2 Loss Function 21
2.3.3 Gradient Descent 22
2.3.4 Hyperparameters 22
2.3.5 Imbalanced Dataset 24
2.3.6 Other Applications of ML in compilers 24

2.4 Neural Networks 25
2.4.1 Semantical Representations 25
2.4.2 Activation Functions 26
2.4.3 Layers 26
2.4.4 Siamese Neural Network 29

3 Design, Methodology & Implementation 31
3.1 General Design and Methodology 31
3.2 Data Collection 32
3.2.1 Use of IR2Vec 33
3.2.2 Database Solution 33
3.2.3 Database Design 33
3.2.4 Data Collection Framework 35
3.3 Model Development 36
3.3.1 Data Pre-Processing 36
3.3.2 ML Model Design 37
3.3.3 Hyperparameter Tuning 39
3.3.4 Framework/Pipeline for Model Development 41
3.4 System Integration 42
3.4.1 Integrating Tensorflow 42
3.4.2 Integrating IR2Vec 43
3.4.3 How it all fits together 43
3.5 Artifactory and Set up Script 44
3.5.1 Repository and Accessibility 44
3.5.2 Automated Setup Script 44

4 Evaluation 45
4.1 Experimental Set Up 45
4.2 ML Model Prediction Accuracy 45
4.2.1 Dot Product or Cosine Similarity 46
4.2.2 Dot Product Siamese Model Results 47
4.2.3 Multi-Headed Self-Attention Model Results 48
4.2.4 Evaluation 48
4.3 Quality of Merging Predictions 50
4.3.1 Number of Attempted Merges 50
4.3.2 Profitability of Attempted Merges 52

4.3.3 Evaluation 52
4.4 Code Size Reduction 52
4.4.1 Compiled-Code Segment Size Reduction 54
4.4.2 Binary Size Reduction 54
4.4.3 Evaluation 57

5 Conclusion **59**
5.1 Summary 59
5.2 Achievements 60
5.3 Reflection 61
5.4 Future Work 62

Bibliography **64**

A Appendix **69**
A.1 Acronyms 69
A.2 Calculations 69
A.3 Artifacts 69

Word Count: 0

List of Figures

2.1 The MinHash algorithm: Textual documents are broken up into overlapping shingles; Each shingle is hashed using *k* hash functions, retaining only the smallest hash for every function, producing a fingerprint of *k* elements. Figure taken from [47]. 18
2.2 **Locality Sensitive Hashing** - The fingerprint on the left side represents a candidate function, and the pairing fingerprint on the right side represents potential functions to merge with the candidate. The pairing fingerprints share at least one band with the candidate fingerprint, so they will be considered during candidate lookup. Figure taken from [47]. 19
2.3 **Sequence Alignment between Two Functions**, green nodes represent equivalent instructions and are aligned next to each other, and red nodes represent un-equivalent instructions, placed next to the gaps. Figure taken from [40]. 20
2.4 Graph of sigmoid activation function on the left and the ReLU activation on the right. Figure taken from [54]. 27

3.1 End-to-end pipeline: From data collection of merging performance, through ML model development and evaluation, to final component integration to produce a working function merging pass. 32
3.2 Schema of database used to store collected data using crows foot notation 34
3.3 **Visual representation of the Dot Product Siamese Model.** The encoders for both models share the same weights to map both functions' features onto the same function space. 38
3.4 **Visual representation of the Multi-Headed Self-Attention Model.** The model uses the attention mechanism to allow each element to attend to another. 40

4.1 **Scatter Plot of Cosine Similarity Siamese Model's Predictions vs. True Labels** using 80 million training samples. The diagonal line represents perfect agreement. Scatter plots effectively reveal patterns in smaller datasets; however, they lose clarity as datasets grow larger. In the subsequent sections, the volume of data is significantly larger, so heat maps were used to capture density and trends better. 46
4.2 **Scatter Plot of Dot Product Siamese Model's Predictions vs. True Labels** using 80 million training samples. The diagonal line represents perfect agreement. Scatter plots effectively reveal patterns in smaller datasets; however, they lose clarity as datasets grow larger. In the subsequent sections, the volume of data is significantly larger, so heat maps were used to capture density and trends better. 47
4.3 **Frequency Heat-map of Dot Product Siamese Model's Predictions vs. True Labels.** The colour intensity (log-scaled count) represents the frequency in each prediction–actual score bin, and the diagonal line represents perfect agreement. 48
4.4 **Frequency Heat-map of Multi-Headed Self-Attention Model's Predictions vs. True Labels.** The colour intensity (log-scaled count) represents the frequency in each prediction–actual score bin, and the diagonal line represents perfect agreement. 49
4.5 **Number of Attempted Merges** for each threshold. 51
4.6 **Percentage of merge attempts that are profitable** for each threshold. This graph illustrates the proportion of merge attempts that result in smaller function size calculated using equation A.1. 53
4.7 **Dot Text Size Reductions Across Different Thresholds.** The percentage of dot text size reduction is shown against the baseline, which is the default LLVM without any experimental function merging (Higher is better). 55
4.8 **Binary Size Reductions Across Different Thresholds.** The percentage of binary size reduction is shown against the baseline, which is the default LLVM without any experimental function merging (Higher is better). 56

Abstract

Code size reduction remains critical across computing devices despite increasing hardware capabilities, particularly for embedded systems and mobile applications where memory constraints impact functionality and costs. Function merging addresses this challenge by combining similar functions to eliminate redundancy, but identifying optimal function pairs to merge remains an ongoing research field. This paper proposes using machine learning to improve function merging decisions, replacing the handcrafted heuristics used in previous state-of-the-art implementations like F3M with models capable of capturing complex relationships between functions. Our approach focuses on the alignment score as the primary metric for assessing function similarity, as it effectively quantifies structural similarity for merging. A robust data collection framework was developed to gather information on 2.2 billion function pair merges and their performance across diverse benchmarks. Two neural network architectures, a dot product Siamese model and a multi-headed self-attention model, were designed and trained using the collected dataset to predict alignment scores between function pairs. F3M was modified to leverage these trained models by predicting the alignment score between a function and all candidate functions. Merging is only attempted with the highest-scoring candidate whose score exceeds a predetermined threshold. Evaluation on SPEC CPU 2006 and 2017 benchmarks demonstrates that our approach improves F3M's compiled-code segment size reduction (the portion of the binary containing actual machine instructions over which function merging has direct control) by 48% (4.4% versus 2.9% reduction). While the models achieve slightly lower overall binary size reduction compared to F3M due to increased exception handling metadata required for the complex control flow of merged functions, they exhibit significantly more reliable behaviour, rarely producing binaries larger than the baseline LLVM's compilation, a notable improvement over F3M's occasional size increases.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my supervisor, Dr. Pavlos Petoumenos, for his invaluable guidance and support throughout this project. His insightful advice, constructive feedback, and innovative ideas have significantly shaped my approach and this research's outcomes. His patience and encouragement in my academic growth have been instrumental in helping me navigate the complexities of this work. I sincerely appreciate the time he dedicated to our discussions and creating an environment where I felt comfortable exploring new concepts and techniques.

I extend my heartfelt thanks to my family for their unwavering support and for providing me with the opportunity to pursue this work. I am also grateful to my friends, whose valuable advice regarding project selection and consistent encouragement helped me maintain my motivation and perspective. Your collective support has been invaluable.

Chapter 1

Introduction

1.1 Motivation

It is hard to imagine the benefits of minimising code size in the current technological environment, where many developers de-prioritise it in favour of additional features and quality due to the abundance of memory and processing power. Moore's law states that the number of transistors on an integrated circuit doubles every two years and has held for decades [32]. However, code size reduction remains a critical concern across the computing spectrum, from small, interconnected Internet of Things (IoT) computing devices embedded in everyday objects to data centres. While often overlooked in favour of performance optimisations, code size can become a constraint in many scenarios. This is particularly true for resource-constrained devices where memory limitations directly impact functionality, cost, and adoption.

Code size remains a primary concern in embedded systems, where memory and computation power are severely limited. As our surroundings become more digitised, more embedded systems will be around us, such as cameras, home control systems, and various IoT devices. Memory typically occupies the most significant fraction of chip area in these systems, contributing significantly to overall manufacturing costs [33]. Even small increases in memory requirements translate directly to equivalent cost increases, which can lead to substantial cost increases at scale [53]. Generating smaller binaries can relieve the stress on hardware specifications needed to host embedded software, decreasing costs and lowering e-waste once these devices reach their end-of-life.

In the mobile sector, smaller binaries offer several advantages. Faster loading times contribute to a more responsive user experience, extending the device's lifecycle and enhancing overall satisfaction. Mobile operating system vendors limit the size of binaries that can be downloaded over mobile data to avoid excessive costs and prolonged wait times for users. Suppose an application's size surpasses the threshold. The number of downloads will decrease, as users tend to install applications when needed but seek different solutions when faced with barriers [19]. Application delivery platforms further constrain executable size, with Google Play not allowing compressed APKs larger than 200MB and Apple's App Store capping executables at 500MB [1][11].

Function merging is a promising approach for reducing the binary size by eliminating code redundancy. This technique works by identifying functions with similar code structures and consolidating them into a single function while preserving the functionality of the original functions. The merged function contains the union of the original functions' behaviour, with conditional logic to handle function-specific paths. However, identifying which function pairs to merge presents significant challenges, as merging dissimilar functions may introduce additional instructions (e.g., branches) to handle the differences between merged functions, potentially making the merge unprofitable.

These optimisations are especially valuable for modern programming paradigms. High-level abstractions in languages like C++ often introduce duplicate code through templates, multiple constructors/destructors, and other specialisations [48]. Function merging makes these abstractions more practical by eliminating the resulting code redundancy. Unfortunately, production compilers offer limited support for advanced function merging, typically only combining perfectly identical functions [7], with further experimental improvement to merge functions with identical control-flow graphs [23]. Research compilers have extended these capabilities by using fingerprints and hashes to identify more viable function pairs to merge [40, 47]. However, these current state-of-the-art systems use handwritten heuristics, like FMSA's fingerprint-based and F3M's hash-based similarity metrics, to estimate the similarity between function pairs. These approaches can lead to missed merging opportunities, particularly in specialised cases.

In contrast, a machine learning approach has the potential to automatically learn to predict function pair similarity more accurately by capturing complex patterns that determine which functions can be profitably merged rather than relying on hand-crafted metrics. On multiple occasions, machine learning and deep learning have outperformed human-designed heuristics when designing code-related heuristics [21, 51, 46]. By analysing patterns across millions of function pairs, ML models can identify subtle indicators of similarity that might be missed through human observation.

1.2 Aims

This project aims to develop and evaluate a machine learning-based approach to improve function merging optimisation by replacing the hand-designed function pair similarity heuristic with a deep learning model that predicts the score.

To achieve these aims, this project encompasses the following objectives:

1. Design a framework to collect function pair similarity scores across multiple benchmarks, implementing efficient storage solutions to store 2.2 billion function pairs' data.

2. Design and train a deep learning model that can reliably predict similarity scores between a pair of functions with high accuracy.

3. Integrate the trained ML model into an LLVM function merging pass for better merging-decision heuristics to create a better function merging pass.

This project's achievements will be evaluated in three folds: (1) the trained model's ability to correctly predict how similar two functions are, (2) the system's performance at identifying suitable candidates for merging by translating the prediction scores into merging decisions, and (3) the overall code size reduction compared to both production compilers and current state-of-the-art implementation.

1.3 Report Structure

The rest of the paper is structured as follows:

- **Chapter 2** - Introduces the LLVM compiler infrastructure project in section 2.1. Section 2.2 examines different implementations of function merging, including previous state-of-the-art realisations. Sections 2.3 and 2.4 cover this project's machine learning foundations, including exploring other applications of machine learning to compilers.
- **Chapter 3** - Details the design and methodology decisions this project employs to meet the project objectives. Section 3.1 discusses the overall design of the project, section 3.2 discusses designs of the data collection process, section 3.3 goes through the design of the models and techniques employed for training the models. Section 3.4 integrates the trained models into the function merging pass and 3.5 talks about the setup script and artifacts provided by this project.
- **Chapter 4** - Describes the environment used to evaluate this project and evaluates this project as discussed in section 1.2. The machine learning models' accuracy is assessed in section 4.2, the quality of the merges is then evaluated in section 4.3 and finally, the overall code size reduction is discussed in section 4.4.
- **Chapter 5** - Concludes the whole project, summarising the work carried out throughout the whole project in section 5.1, discusses this project's achievements in section 5.2, reflects on the work done in section 5.3 and suggest future works in section 5.4.

Chapter 2

Background

Function merging is a powerful code optimisation technique that reduces redundancy by combining similar or identical functions within a program. At its core, function merging involves identifying functions with similar structures and semantics and generating a merged function that uses decision points and parameters to distinguish between the behaviours where the original functions diverge.

2.1 The LLVM Compiler

LLVM is an open-source compiler infrastructure project widely used in industry and academia that provides a collection of modular and reusable compiler and toolchain technologies [3].

This project is built upon a state-of-the-art (SOTA) implementation of function merging, F3M [47], which was implemented in LLVM. The modular pass-based architecture of LLVM (section 2.1.3) makes it ideal for integrating new optimisations into the pipeline without too much complexity. Additionally, LLVM is widely adopted across industry and academia, with support for many high-level languages and target architectures, ensuring that our optimisation can have a broad practical impact.

2.1.1 LLVM Architecture

At its core, the LLVM compiler employs a three-phase design. The **front-end** translates the source code into LLVM Intermediate Representation (LLVM IR), the **middle-end** performs various transformations and optimisations on the IR, and the **back-end** generates the machine code for specific target architectures from the IR.

This modular design reduces the implementation effort compared to directly porting each language to each target architecture and it also allows LLVM to support multiple programming languages across multiple target architectures while sharing the middle optimisation layer.

2.1.2 LLVM IR

LLVM IR serves as a central data structure throughout the compilation process. It is a language-independent, low-level representation designed to support program analysis and transformation. The LLVM IR comes in three flavours: a human-readable assembly-like text format, an in-memory data structure used by compilers, and a dense bitcode format for disk storage [4].

LLVM IR uses a static single assignment (SSA) form, where each variable is assigned exactly once, making data flow analysis more straightforward. The IR represents programs as a collection of functions, with each function containing basic blocks connected by control flow graphs. Each basic block consists of a sequence of instructions that execute sequentially.

2.1.3 LLVM Passes

LLVM's optimisation framework operates on the IR through a series of passes which analyse or transform the program in steps. Passes are split into three types, the analysis passes which gather information to be used by other passes or the debugger, the transform passes which modify the program in some way and the utility passes which encompass any passes that do not fit into the first two categories [6].

The function merging pass described in this paper is located in the middle-end of LLVM, operating on the IR during link time optimisation (LTO). LTO provides the compiler with global visibility and access to all functions across the program, substantially increasing opportunities for identifying and combining similar code by increasing the pool of function candidates [40].

2.2 Existing Function Merging Approaches

This chapter will discuss the current function merging technique used in LLVM and the previous state-of-the-art solutions to function merging. Function merging is still an actively researched optimisation in compilers, focussing on the orthogonal problems of identifying functions that merge well and how to merge a specific pair of functions.

2.2.1 LLVM Function Merging

The current LLVM implementation adopts a conservative two-phase approach to function merging, prioritising low compilation overhead over maximising merge opportunities, focussing primarily on merging structurally identical functions through an efficient filtering mechanism.

Hashing

The first phase categorises functions using a lightweight hash-based approach to identify potential candidates quickly. The hash aims to capture the function's structural semantics while maintaining the invariant:

$$F1 == F2 \Rightarrow Hash(F1) == Hash(F2) \tag{2.1}$$

It is, therefore, guaranteed that if two functions are the same, they will hash to the same value. This hash function accounts for function signatures, control flow structure, and instruction opcodes. Only functions producing identical hash values proceed to the next phase for a more detailed comparison.

Detailed Comparison

In the second phase, a detailed comparison is applied to functions with matching hash values to verify their structural compatibility for merging. This process begins by checking the functions' attributes, return types and parameter types to ensure type consistency. Then, a comprehensive analysis of the function bodies is conducted. This is done by traversing both functions in parallel using their control-flow graph (CFG), visiting every basic block in the graph, and comparing the instructions inside each basic block. Every instruction in the first function is directly compared to the instruction in the second function in the same position, comparing the instruction opcodes and operand types. When a successful match is identified, LLVM creates a unified function that preserves the behaviour of both original functions [5].

Limitations

This existing approach is inherently conservative, primarily targeting functions with nearly identical control flow graphs and instruction sequences [7]. While it accommodates cases where operand types differ but can be safely bitcasted, this flexibility is quite limited. This conservative stance significantly limits merging opportunities, particularly for functions that contain substantial shared code but differ in specific regions.

These limitations establish a clear opportunity for more sophisticated merging decision mechanisms that recognise non-obvious structural similarities and better predict optimisation benefits.

2.2.2 F3M: Fast Focused Function Merging

Fast Focused Function Merging (F3M) represents the current state-of-the-art function merging technology to address inefficiencies that previously made more aggressive function merging impractical for large-scale applications. This project is built on F3M's infrastructure.

Merging two functions is computationally expensive, attempting to merge every possible function pair is prohibitively impractical. Therefore, function merging implementations rely on an ordering scheme to prioritise the most promising merges. Each function is associated with a fingerprint that numerically represents its key semantic characteristics, particularly those most relevant for identifying similarities between functions. During the selection process, the similarity score is calculated between a candidate function's fingerprint and all other functions in the codebase. Function pairs with the highest similarity scores are evaluated for merge feasibility and profitability.

In general, function merging can be broken down into two main stages, function selection and function merging. Function selection is the process of finding the most promising function pairs to merge, and function merging is the process of generating a merged function from two functions. Since this project mainly focuses on function selection, which is orthogonal to the task of merging the selected functions, this project will reuse the function merging implementation used by F3M.

In F3M, the function selection process involves three stages, MinHash fingerprint generation for every function, locality sensitive hashing to efficiently narrow down the best candidate functions for merging and a similarity calculation between the promising function pairs to determine the best function to merge with. Once the selection is complete, the function merging process involves aligning instruction subsequences and a code generation phase that transforms the alignment information into a new merged function. Each of these steps is further discussed in the following subsections.

MinHash Fingerprint Generation

The ***Jaccard index*** is an ideal similarity metric for identifying potential function merging candidates because it effectively captures the instruction subsequences that can be aligned between two functions. The Jaccard index between two functions, A and B as two sets of instructions are given:

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

Unfortunately, computing the Jaccard index directly has linear complexity relative to the size of the function sets, making it computationally infeasible for practical use. Instead, F3M uses ***MinHash*** as an efficient approximation of the Jaccard index by fixing the size of the function's fingerprints [17].

First, each function's fingerprint based on MinHash needs to be generated. Each instruction is first encoded as a 32-bit integer incorporating four critical properties: opcode, result type, number of operands, and operand types. This encoding abstracts away textual differences between instructions that have the same semantics (such as different operand names with identical opcodes), focusing only on properties relevant to merging.

These encoded instructions are then grouped into overlapping ”shingles” (pairs of consecutive instructions). Multiple hash functions are applied to these shingles; for each hash function, only the minimum hash value across all shingles is retained. This process produces a fixed-length fingerprint vector for each function, with the length corresponding to the number of hash functions used.

In the MinHash fingerprint, each hash value approximately represents a randomly selected shingle from the function. When identical sets of hash values appear in two

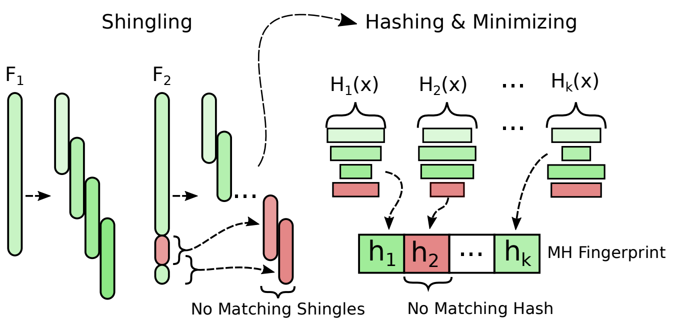


Figure 2.1: The MinHash algorithm: Textual documents are broken up into overlapping shingles; Each shingle is hashed using k hash functions, retaining only the smallest hash for every function, producing a fingerprint of k elements. Figure taken from [47].

different fingerprints, it strongly indicates that both functions contain the same instruction sequence at some point. Conversely, differing hash values suggest a reduced likelihood that the functions share that particular code pattern. This property allows MinHash to efficiently estimate the structural similarity between functions without direct instruction-by-instruction comparison.

Comparing the Fingerprints

After generating the fingerprints for the functions, each function needs to look for another function which it most likely aligns with using the fingerprints. The first step is to find a way to compare the fingerprints and quantify their similarity. This can be done by calculating the Jaccard's index between two functions' fingerprints, which is the ratio of identical pairwise hash values ($A \cup B$) to the number of hash functions (k):

$$J(A,B) = \frac{|A \cap B|}{k}$$

One approach is to find the best candidate function for each function by exhaustively calculating the Jaccard's index for every function and merging the candidate with the highest index score. Despite the complexity of the index calculation having been lowered from linear to constant due to the fixed size of the fingerprints, this can still be expensive.

Locality Sensitive Hashing (LSH)

F3M employs ***Locality Sensitive Hashing (LSH)*** as a more efficient nearest neighbour searching approach by pruning the search space [22]. LSH works by splitting the fingerprint vector into b non-overlapping sub-vectors. Each sub-vector is hashed to produce b values or *bands*. When two fingerprints are similar, they will share at least one band, while dissimilar fingerprints tend not to share any bands at all. During the searching process, the nearest functions can be determined by looking at any other functions that share at least one band with the current function. This process decreases the number of functions to search through to find the most viable candidate, speeding up the search for candidate functions to merge with. This can be visualised in figure 2.2

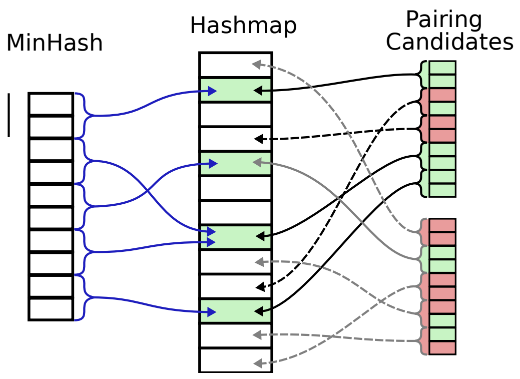


Figure 2.2: Locality Sensitive Hashing - The fingerprint on the left side represents a candidate function, and the pairing fingerprint on the right side represents potential functions to merge with the candidate. The pairing fingerprints share at least one band with the candidate fingerprint, so they will be considered during candidate lookup. Figure taken from [47].

Merging Functions

Since F3M mainly focuses on improving the function selection process, it reuses the code generator used by HyFM [39]. Code generation involves two steps: aligning instruction subsequences and generating a unified code from the alignment.

Instruction Alignment Function merging requires identifying similar instruction sequences that can be combined, and sequences that differ and require special handling. HyFM introduced, and F3M adopted, a more efficient approach by aligning at the **basic-block level** rather than function level. This significantly reduces computational overhead since basic blocks are typically much shorter than entire functions, mitigating the impact of alignment algorithms' complexity. To align instructions, we must first define what makes instructions match. Two instructions are considered matching if they can be merged to produce a single instruction. F3M offers two complementary alignment methods with different trade-offs between speed and flexibility:

Needleman-Wunsch Alignment (NW) For blocks with different sizes, F3M employs the NW algorithm originally introduced to function merging by FMSA [40]. This dynamic programming approach, borrowed from bioinformatics, where it identifies similarities between protein sequences, can optimise alignments between instruction sequences of different lengths [34]. While this NW-based approach can align instruction matches even when instructions appear at different relative positions in their respective functions, this algorithm has quadratic time and space complexity relative to the sequence length. This approach works by inserting gaps in both sequences to create equal-length sequences with optimal alignment, where matching instructions align. In contrast, non-matching instructions align with gaps, demonstrated in figure 2.3.

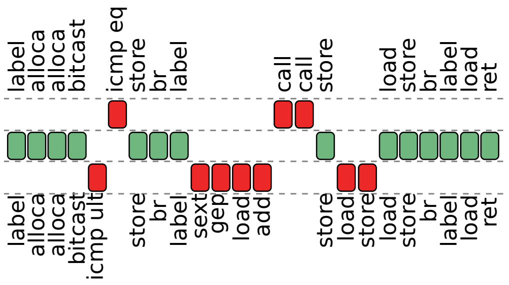


Figure 2.3: Sequence Alignment between Two Functions, green nodes represent equivalent instructions and are aligned next to each other, and red nodes represent un-equivalent instructions, placed next to the gaps. Figure taken from [40].

Pairwise Alignment (PA) HyFM observed that most profitably merged basic blocks have highly similar structures [39]. PA exploits this observation by only attempting to match instructions at corresponding positions in blocks of equal size. This strict positional alignment has linear time complexity, providing a speed-up over the NW alignment.

Alignment Score The alignment between two functions is a property that identifies the structural similarity between two functions, serving as an early predictor of function merging profitability and is the characteristic our machine learning model aims to predict. To quantify this characteristic, the alignment score is defined as the ratio of aligned instructions to the union of aligned (A) and unaligned (U) instructions:

$$Alignment\ Score = \frac{|A|}{|A \cup U|}$$

Using figure 2.3, we can calculate the alignment score as follows:

$$\frac{No.\ of\ Green\ Nodes}{Total\ No.\ of\ Green\ and\ Red\ Nodes} = \frac{14}{24} = 0.5833$$

Given that this metric is a ratio, this metric will be a continuous value in the range of [0, 1] inclusive, where 0 represents an entirely dissimilar function pair, and 1 represents a structurally identical function pair.

Code Generation From Alignment Once the instruction alignment is complete, F3M uses the alignment results to generate a unified function that combines the functionality of both input functions. This process follows HyFM's code generation approach. The merged functions are constructed with an additional parameter, the function identifier, which controls the original function's behaviour to execute. The code generator processes the alignment as follows:

Handling Matched vs. Unmatched Code A single instruction is generated for matching instruction pairs that execute unconditionally in the merged function to serve both original functions. For non-matching instructions, the code generator includes them in the merged function by guarding their execution within conditional branches based on the function identifier. This ensures that the merged function preserves the exact semantics of both original functions.

Control Flow Graph (CFG) Construction The CFG of a function represents the flow of the function, where a node corresponds to a basic block and each directed edge corresponds to a possible transfer of control (e.g. branch) between the blocks, capturing all the execution path through the function [25]. The CFG of a merged function is constructed by breaking the original basic blocks into smaller ones that separate matching from non-matching code.

2.3 Machine Learning

Machine learning (ML) offers an alternative by shifting from explicitly programmed rules to data-driven decision models. The fundamental advantage of ML-based approaches lies in their ability to discover and leverage intricate patterns across multiple code structures and behaviour dimensions.

2.3.1 Supervised Learning

Supervised learning is a machine learning paradigm in which learning is guided by labelled data [43] [10]. In this approach, each input example is paired with a corresponding output, allowing the model to learn a mapping from inputs to outputs by minimising a defined loss function. This process often involves iterative optimisation techniques to adjust model parameters and improve prediction accuracy.

In our context, supervised learning enables a model to predict the similarity of two functions by training on examples of previously merged functions and their alignment score.

2.3.2 Loss Function

A loss function is a mathematical function that quantifies the difference between the model's current predicted value and the ground truth.

Loss functions fall into two primary categories corresponding to the tasks, regression models (predicting from a continuous spectrum) and classification models (predicting from a discrete set of values). This paper will focus on regression loss functions to predict the alignment score (section 2.2.2) for a pair of functions.

Mean Squared Error (MSE)

Mean squared error is a widely used loss function for regression tasks, defined by the following formula:

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

y_i = Ground Truth Value
$\hat{y}_i$ = Predicted Value
n = Number of Samples

The squaring operation in MSE provides two benefits. First, it makes all errors positive, ensuring that when multiple errors are combined and averaged in a batch, positive and negative errors do not cancel each other out. Second, it emphasises larger errors due to the quadratic nature of the function, prioritising the reduction of substantial prediction errors before fine-tuning smaller discrepancies, a desirable property when dealing with small values in the range of 0-1.

A smaller MSE is better, signifying reduced discrepancy between the predicted value and the ground truth. For our function merging application, MSE is preferable to alternatives like Mean Absolute Error (MAE) because the quadratic penalty helps the model focus on avoiding large prediction errors that could lead to poor merging decisions.

2.3.3 Gradient Descent

Gradient descent, the primary optimisation algorithm in deep learning, uses forward and backward propagation. During **forward propagation**, the model processes the training data through itself to produce a set of predictions. The loss function quantifies the error between these predictions and the ground truth.

During **backward propagation**, the gradient for the loss function is calculated with respect to each parameter in the model using the chain rule. The gradients indicate the direction and magnitude of parameter adjustment needed to reduce the loss [42, 29].

Using this information, gradient descent adjusts each parameter in the opposite direction of the gradient, scaled by the selected learning rate [14]. This process iteratively navigates the parameter space toward a local minima of the loss function if it is well designed. Finding a global minima is not always possible, so a well-suited local minima is sufficient for predictions.

All gradient descent methods aim to minimise the loss function, iteratively refining the model to make predictions that more closely match the ground truth.

2.3.4 Hyperparameters

An ML model usually has hyperparameters to configure. These model parameters are not learnt and configured before a model starts training. Unlike model weights, which are optimised during training, hyperparameters control the learning process itself. They control how the model learns and behaves from the input data. A subset of the dataset, the validation set, is usually set aside to tune the hyperparameters. Some of the most common hyperparameters are discussed below.

Epochs The number of epochs specifies the number of times the model trains using the entire training data. Setting a value that is too low may cause the model to underfit[1] to the training data, while a value that is too high may cause the model to overfit[2] to the training data, causing a drop in performance when evaluated against the validation set. One way to prevent this type of overfitting is by implementing early stopping. Early stopping monitors the model's performance on the validation data, halting training when the validation performance degrades, preventing overfitting.

Learning Rate The learning rate determines the size of the step taken by the model for each training iteration, a large learning rate would magnify the loss function more than a smaller learning rate. Selecting a large value means less time is needed for training, but the model may overshoot the global minimum and struggle to converge. On the other hand, if the learning rate is too small, the model will converge on a local minimum and be unable to leave it, thus not finding the optimal solution [15]. Adaptive learning rate techniques like Adam can adjust the learning rate dynamically during training to improve convergence [41].

Batch Size The batch size specifies the number of training samples processed in each training iteration. The loss and gradient are averaged for all training samples in a batch. Selecting a suitable batch size is important. One too small produces noisy gradient estimates, leading to unstable weight updates, hampering convergence toward the true minimum. At the same time, one that's too large requires more memory and may smooth over important gradient details, potentially causing convergence to a sub-optimal solution [27, 38, 14].

Hyperparameter Optimisation Hyperparameter optimisation is fundamentally an iterative process of proposing hyperparameter settings, training the model and evaluating its performance on a validation dataset. The hyperparameter settings are refined until the optimal validation performance is reached. **Grid search** exhaustively samples this space but becomes inefficient as the dimensionality grows [9]. **Bayesian optimisation** is an alternative which builds a probabilistic model of the model's performance on the validation set to suggest promising hyperparameters effectively [52]. At each step, Bayesian optimisation maintains a belief about how well different hyperparameter settings will perform, then uses the results of past trials to update that belief using Bayes Theorem [13]. This updated belief is used to influence the following settings to

[1]**Underfitting** - When a model is unable to capture the underlying patterns in the training data, leading to poor performance on training and unseen data

[2]**Overfitting** - When a model starts memorising the training data, capturing noise and irrelevant information instead of trying to learn the patterns associated with the data leading to poor generalisation on unseen data

try, balancing the desire to explore uncertain regions against the chance of finding an even better result. In this way, Bayesian optimisation focuses effort where it is most likely to pay off, often finding high-performance configurations in fewer trials than grid search [9].

2.3.5 Imbalanced Dataset

In a perfect world, a dataset will have a similar amount of data samples across the output data's range/spectrum, but this is not always possible. If no effort is taken to take into account the imbalance of the data samples, a model will learn to predict more of the dominant data samples. To mitigate this, sample weightings can be used, which will scale down the loss for the dominant data points and scale up the loss for under-represented data points. This is known as **cost-sensitive learning (weighting)** [37]. This ensures that the loss function will reasonably adjust the parameter without overfitting itself to predict more for the dominant data points.

2.3.6 Other Applications of ML in compilers

Optimisation Selection Optimisation Selection focuses on determining which compiler optimisations to apply to a program to enhance performance, reduce code size, or meet other objectives. Traditionally, compilers have relied on fixed rule-based heuristics developed by compiler engineers to choose from a vast array of optimisation passes. However, with techniques like those used in MILEPOST GCC, machine learning models now collect program features and runtime feedback to automatically predict the most beneficial optimisation strategies, thereby replacing static rules with adaptable, data-driven decisions [24].

Iterative Compilation Iterative Compilation tackles the optimisation challenge by repeatedly compiling a program with different settings and empirically testing the resulting performance. Instead of depending solely on predefined heuristics, this method explores a broader range of optimisation configurations to identify the optimal solution. The process is enhanced by active learning strategies which intelligently select training examples and adjust sampling dynamically based on the model's uncertainty. This adaptive approach dramatically reduces the time and effort required for extensive fine-tuning [35].

Loop Vectorisation Loop Vectorisation is a compiler optimisation that transforms sequential loop operations to exploit the parallel processing capabilities of SIMD hardware. Traditional vectorisers use fixed-cost heuristic models that may not fully capture the complex data dependencies and instruction patterns within code loops. In contrast, advanced approaches like NeuroVectorizer use deep reinforcement learning to predict the best vectorisation factors for loops. This method learns code representations directly from the source code, allowing it to determine more effective vectorisation and interleaving factors [26].

Function Inlining Function Inlining is a technique that replaces a function call with the body of the function itself, thereby eliminating the overhead associated with calling a function. Recent advances, such as MLGOPerf, extend machine learning techniques to optimise inlining decisions. This approach leverages a two-model strategy: a primary reinforcement learning model for decision making and a secondary model to predict the performance gains from inlining. By doing so, the system efficiently optimises inlining, and has demonstrated improvements in performance on standard benchmarks compared to traditional inlining strategies [12].

2.4 Neural Networks

Neural networks are powerful machine learning models inspired by the structure and function of the human brain. They automatically learn hierarchical representations from raw inputs through **artificial neurons** arranged in an input layer, hidden layers, and an output layer. Each neuron receives input signals, applies weights and biases, and passes the result through a non-linear activation function to produce an output. A network's performance is influenced by its **depth** (number of layers) and **width** (neurons per layer). Training involves optimisation algorithms like gradient descent and back-propagation to adjust weights and minimise a loss function.

2.4.1 Semantical Representations

Machine learning models face a fundamental limitation when processing data, they operate exclusively on numerical values. This creates a significant challenge when working with symbolic or textual data, such as program code. Since these algorithms rely on mathematical operations, inputs must be represented numerically.

To overcome this limitation, inputs should be transformed into numerical representations that preserve semantic relationships. Word embeddings exemplify this approach by mapping words into dense, continuous vector spaces that capture semantic information.

Simple encoding techniques like **one-hot encoding** fail to capture meaningful relationships between elements because all symbols are equidistant, resulting in inefficient high-dimensional, sparse representations that increase computational costs.

More advanced embedding techniques (word2vec and GloVe) enable machine learning models to effectively utilise the underlying semantics of symbolic data for prediction tasks [31, 36].

IR2Vec

IR2Vec is a framework that generates embeddings for LLVM's Intermediate Representation (LLVM IR) [50]. Unlike traditional word embeddings, IR2Vec transforms programs and functions into vector representations by modelling the relationships among IR entities, such as opcodes, types, and operands, using a translational embedding model like *TransE* [16].

The process extracts relational information from LLVM IR as triplets (¡h, r, t¿), representing relationships between entities. Using these triplets, IR2Vec builds vector representations of the IR's fundamental components (opcodes, operands, etc.) as seed encoding. These seed embeddings then construct higher-level embeddings for instructions, basic blocks, functions and modules (Symbolic Encoding).

IR2Vec offers an even richer embedding by incorporating the program's flow details, like use-def chains and reaching definitions, into the symbolic encoding (flow-aware encoding), capturing both operational semantics and syntactical information for more effective analyses and optimisations.

IR2Vec is fully open-source, facilitating straightforward customisation and extension. It is available across multiple platforms, it can be utilised as a Python library, a C++ library, or even as a stand-alone binary, ensuring seamless integration into various development environments.

2.4.2 Activation Functions

Activation functions are mathematical operations applied to the output of neural network layers that introduce non-linearity, allowing networks to learn complex patterns beyond what linear combinations can express. The ReLU (Rectified Linear Unit) activation function has become the standard choice for hidden layers, outputs a value in the range $[0, \infty)$, passing through positive inputs unchanged while zeroing out negative values. In contrast, the Sigmoid activation function squashes inputs to outputs between 0 and 1, making it well-suited for models that need values to be normalised within a bounded range. Figure 2.4 shows the graphs of the sigmoid and ReLU activation functions.

2.4.3 Layers

This section will discuss the layers and techniques used in the paper.

Dense Layers

Dense or fully connected layers are a fundamental building block of neural networks. In a dense layer, every artificial neuron is connected to every neuron in the preceding layer, creating a network of weighted connections [14]. The number of neurons in a dense layer could also be treated as a hyperparameter to be tuned. A dense layer

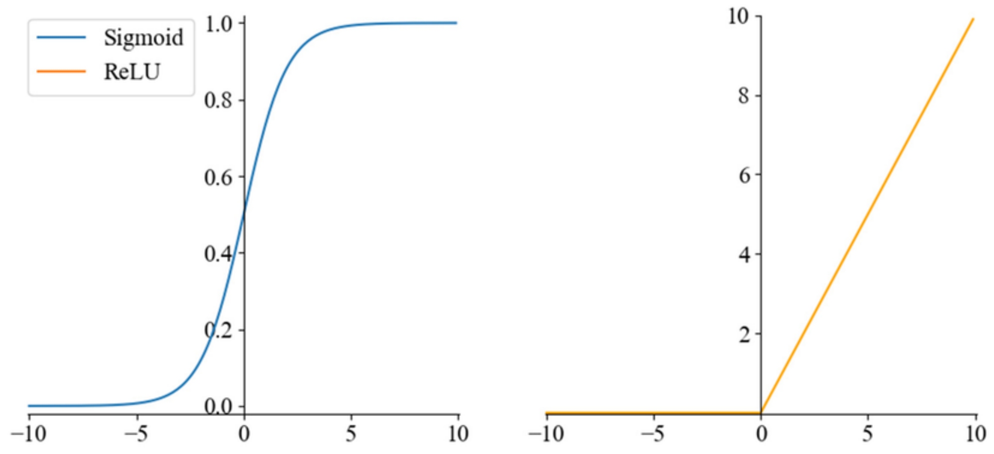


Figure 2.4: Graph of sigmoid activation function on the left and the ReLU activation on the right. Figure taken from [54].

performs the following mathematical transformation:

$$y = \sigma(Wx + b)$$

$x =$ Input vector from the previous layer
$W =$ Weights matrix
$b =$ Bias vector
$\sigma =$ An activation function to introduce non-linearity

Dense layers are known for their powerful feature extraction abilities by learning complex patterns and relationships within data. They transform an input's representation into a more abstract representation. The primary advantages of dense layers include their flexibility in handling fixed-size vector inputs and their strong representational capacity.

This layer can lead to overfitting in deep learning networks, which is why dense layers are often used with regularisation techniques.

Dropout Layer

Regularisation techniques seek to prevent overfitting by introducing controlled noise during training so that models learn representations that generalise beyond the training set. One of the most popular methods for neural networks is dropout, which directly alters the network's architecture during training instead of the usual modification of the loss function.

At its core, dropout is very simple, each neuron in a layer is randomly deactivated on each training sample with probability p (dropout rate). Equivalently, each neuron is kept with probability $q = 1 - p$. During the forward pass, a binary mask m of independent Bernoulli random variables is sampled and applied to the neurons, any surviving neurons are scaled by $\frac{1}{1-p}$ so that their expected value remains unchanged [45].

Normal neural layer's mathematical representation:

$$y = f(Wx + b)$$

After applying dropout:

$$y = f(Wx + b) \times \mathbf{m}$$

There are two variants of dropout, the standard dropout which applies the scaling during inference, and the inverted dropout which applies the scaling during training. Modern frameworks use inverted dropout by default to simplify inference [8, 2].

Typical choices for the dropout rate p range from 0.2 to 0.5, although these values should be treated as hyperparameters and tuned for each task [45].

Multi-Headed Attention Layers

Attention layers allow a neural network to focus on the most important feature of the current task. Not all parts of the input contribute equally to solving a problem; by learning which elements require more attention, the network can make more informed decisions.

The attention mechanism takes in two inputs, the first input will attend to the second. There are two variants of attentions, **self-attention** where both inputs are the same, so the sequence is attending to itself, or **cross-attention** where both inputs are different.

The attention mechanism starts with three learned weights used to create three vectors from the inputs, the query (Q) vector is created from the first input, and the key (K) and value (V) vectors are created from the second input.

Then, it compares each query against all available keys to find how well each key matches the query by calculating the dot product of the pair. High-scoring pairs are given more attention while low-scoring pairs are given less attention or ignored. The scores are then put through a softmax function that converts each score into a probability. Then, the value vector is multiplied by the weight, and the elements are summed up to generate a weighted sum to produce the first input's new vector representation after attending to the second input [49].

The model then uses the newly attended vector for further processing and prediction. The error is calculated, and backpropagation is then used to update the weights to generate the Q, K and V vectors.

A multi-headed attention layer takes it one step further by having multiple attention layers in parallel to model different relationship aspects. This works by dividing the Q, K and V vector representations across the number of parallel attention mechanisms, each mechanism is known as a head. Each head works on its allocated portion of the Q, K and V values before being concatenated to form the final representation, which is typically projected back to the original dimension.

Despite its effectiveness, attention and multi-headed attention models come with higher computation costs than traditional mechanisms due to the attention mechanism's nature, where every element in the sequence must attend to every other element, resulting in a $O(n^2)$ complexity with respect to the input length [28].

2.4.4 Siamese Neural Network

Siamese neural networks are a class of artificial neural networks that process two inputs in parallel using identical sub-networks with shared weights to produce comparable output embeddings. The embeddings are then compared using a distance or similarity function to produce a value suitable for verification and matching tasks [20].

The architecture's key characteristic is the weight-sharing constraint between sub-networks, which ensures that similar inputs are mapped to nearby points in the embedding space while dissimilar inputs are mapped to distant points. Additionally, this ensures that the embedding is consistent across both inputs, enabling the model to generalise similarity judgements on unseen pairs.

Siamese networks have found wide applications in signature verification, face recognition, and image similarity assessment [18, 44, 30]. Their effectiveness stems from learning a discriminative embedding space rather than direct classification.

Numerous metrics can be employed to measure the similarity between embeddings. In this paper, we focus on the following three:

Manhattan Distance The Manhattan distance, also known as the L1 distance, measures similarity by computing the sum of absolute differences between corresponding elements of two vectors, increasing with dissimilarity. For embeddings a and b of dimension n, the distance is calculated as:

$$d(a,b) = \sum_{i=1}^{n} |a_i - b_i|$$

Dot Product The dot product, also known as the inner product between two vectors a and b, is computed as:

$$\sum_{i=1}^{n} a_i \times b_i$$

When used as a similarity metric in Siamese networks, the dot product measures the directional alignment and magnitude correlation between embeddings. A higher dot product indicates a higher similarity.

Cosine Similarity Cosine similarity is the cosine of the angle between two vectors. It is defined as the normalised dot product of the vectors, removing the influence of vector magnitude and focusing only on the directional similarity. For embeddings a and b, cosine similarity is calculated as:

$$cos(\theta) = \frac{a.b}{|a||b|}$$

The result ranges from -1 to 1, where -1 indicates vectors pointing exactly opposite directions, 0 represents orthogonal vectors with no directional correlation, and 1 signifies vectors pointing in the same direction.

Chapter 3

Design, Methodology & Implementation

This chapter ties the material from the previous chapter to this project by outlining its design, technique, and execution, as well as the rationale behind the choices made along its course.

3.1 General Design and Methodology

Current heuristic-based approaches for identifying function pairs suitable for merging, often miss opportunities for optimisation. This project aims to improve compiler function merging by leveraging machine learning to predict function pairs likely to produce beneficial merges.

For this approach to succeed, comprehensive training data consisting of diverse function pairs and their merging performance must be collected. These functions will be encoded into vector representations using IR2Vec to quantify their semantic meaning. The collected data will be processed to normalise before being used to develop, train and evaluate different ML models. To train the neural network architectures, the model will be fed labelled function pairs consisting of the function's vector representation and the merging performance. Once trained, this model, which predicts alignment scores for new function pairs, will be integrated into F3M to improve the function selection decisions.

The project methodology is structured around three interconnected stages: data collection, machine learning model development, and integrating everything, shown visually by figure 3.1.

The first stage involves collecting a diverse dataset of merging performances to train the neural networks. This involves gathering two essential components: a suitable vector encoding of functions that captures their semantics and a metric that quantifies the similarity between function pairs to determine whether merging would be beneficial.

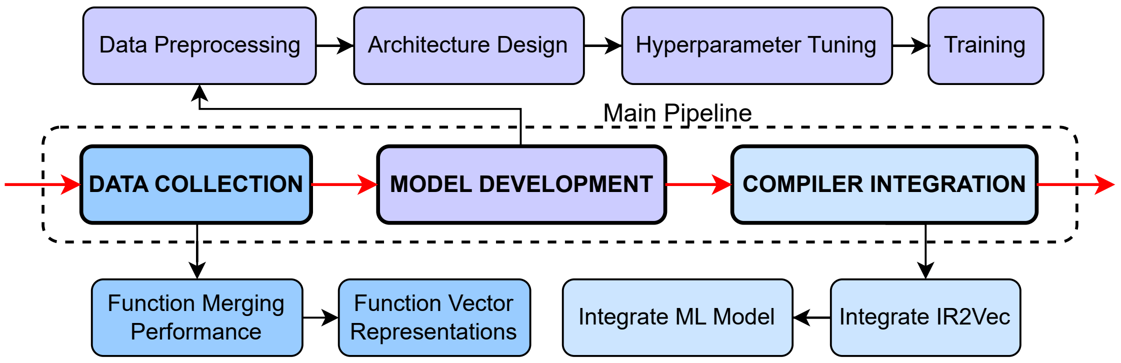


Figure 3.1: End-to-end pipeline: From data collection of merging performance, through ML model development and evaluation, to final component integration to produce a working function merging pass.

In the second stage, various machine learning architectures are explored and fine-tuned using the collected data to evaluate their performance in predicting merge suitability. This experimentation allows us to identify the most effective model architecture for function merging.

In the final stage, the trained models and function encodings are integrated into the LLVM compiler and evaluated on a suite of benchmarks to measure their overall code size reduction performance.

This project also includes artifacts for reproducing results, including a setup script, which will be detailed in the subsequent sections, along with elaborations on each stage of this process, providing insight into the design decisions and implementation details.

3.2 Data Collection

For this project, we gather vector representations of individual functions and pairwise functions' alignment scores, as defined in Section 2.2.2. Since machine learning models perform better with numerical data than raw text, we encode function semantics using IR2Vec (Section 2.4.1). IR2Vec operates directly on LLVM IR and, being open source, can be customised to our needs.

Next, the F3M codebase was extended to interface with a database, storing each function pair's merging performance and outcomes. To create a diverse dataset, we use a representative set of benchmarks that are commonly used in this research area, cc1plus, Chrome, LibreOffice, Linux, LLVM, SPEC2006, and SPEC2007, provided as bitcode files in F3M's artifacts. F3M's function merging mechanism was applied to every possible function pair across these benchmarks to collect their merging performance. While F3M computes multiple metrics, the alignment score was selected as the primary measure as it effectively captures the structural similarity between functions and is important for merging decisions.

Since two sources are used for data collection, a good database design is needed to relate and efficiently store the data accurately.

3.2.1 Use of IR2Vec

IR2Vec (Section 2.4.1) was employed to transform the IR of each function into a 300-dimensional vector embedding. This wide dimensionality allows the encoder to capture more detail for each function. This process also produces an embedding for every function, making it a perfect fit for our needs to work at function-level granularity. Additionally, IR2Vec was selected because of its proven ability to capture both the semantic meaning and flow of LLVM IR.[50]. Its efficient encoding process and graceful handling of out-of-vocabulary tokens via seed embedding vocabulary further contribute to its appeal as an encoder.

3.2.2 Database Solution

Before collecting any data, it was necessary to determine a suitable storage solution that fits the project's needs.

SQL ***SQL*** databases offer significant advantages over other file structures like *CSVs* and *Pandas* primarily due to the expressive power of the SQL query language. Rather than reinventing the wheel each time a complex query is required, SQL provides users with a rich set of built-in functions and operators. This expressiveness simplifies the querying and enhances performance, especially when working with large volumes of data, by leveraging optimised indexing and storage mechanisms. In contrast, Pandas loads the entire dataset into memory, making it impractical for extremely large datasets like we will have. Furthermore, the relational nature of SQL databases, where data is organised into interconnected tables, enables more efficient modifications and queries across related data sets, reducing redundancy and improving maintainability.

SQLite ***SQLite*** was selected over other SQL solutions, such as *PostgreSQL*, primarily for its ease of use and minimal configuration requirements. Unlike PostgreSQL, which typically requires extensive setup, including robust security configurations and server management, SQLite operates as a server-less, file-based database, making it an ideal choice where sensitive data is not a concern. This centralised-file design also simplifies creating a portable artifact, which was a prime consideration in providing a replicable project. Additionally, SQLite provides a C++ interface, the language used to build LLVM and F3M. Finally, its widespread adoption in the Python ecosystem offers an advantage, popular libraries like ***Pandas*** and ***TensorFlow*** provide support to read data from SQLite directly, enabling straightforward connections and interactions.

3.2.3 Database Design

The design of this database schema is guided by the need to efficiently store and query data, especially since the number of function pairs scales quadratically to the number of functions in a program.

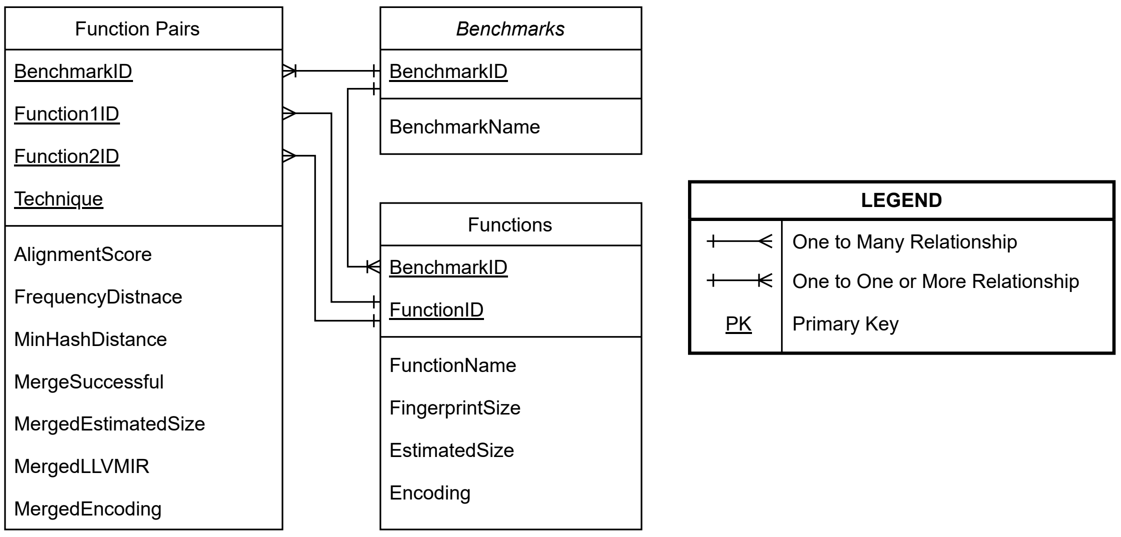


Figure 3.2: Schema of database used to store collected data using crows foot notation

The schema consists of three primary tables, as Figure 3.2 illustrates. The **Benchmarks** table serves as the top-level reference point for all data points. This design ensures that each evaluated dataset is logically isolated, facilitating independent analyses and debugging if needed.

The **Functions** table captures detailed information about individual functions for each benchmark. Combining *BenchmarkID* and *FunctionID* as composite keys ensures that function identifiers are scoped locally within each benchmark, preventing cross-function ID collisions. The foreign keys ensure that each function entry is associated with a valid benchmark.

Each function's name is recorded as a unique key for linking F3M outputs to IR2Vec embeddings. The fingerprint size is computed as the sum of the frequencies of opcodes in HyFM's function fingerprints. Function size is then estimated by summing LLVM's per-instruction code-size costs, queried via the TargetTransformInfo interface.

The functions' encodings are stored as BLOB type to store a pickled Python list representing the function's vector encoding. This decision aims to reduce redundant data transformations since the encodings are generated and used in Python (during training). Storing them directly in binary format avoids unnecessary conversions to and from string representations.

Finally, the **FunctionPairs** table stores all function pair comparisons. The foreign keys ensure that referenced benchmarks and functions exist. This table only uses a single BenchmarkID alongside two function IDs because function merging is only performed within the same program. This table also stores the distance metrics, AlignmentScore (2.2.2), the pairwise fingerprint distance computed by HyFM [39], the pairwise fingerprint distance computed by F3M and any merging outcomes.

3.2.4 Data Collection Framework

The data collection step was split into two steps, one to collect information on F3M's merging attempts and one to collect the functions' embeddings from IR2Vec.

An automated scripts were made for each step, both sharing many similarities. The user would specify a base directory of the benchmarks, and the script would dive into the directory and sub-directories, looking for any benchmark's bitcode files. The scripts have additional parameters to specify the database's stored name and file path, ensuring that different runs do not interact with the same default database. This allows multiple shell sessions to run the same script concurrently on different benchmarks, enabling several benchmark data collections to occur simultaneously. This concurrent execution saves time since each script processes benchmarks sequentially and prevents the corruption of the SQLite file, as SQLite does not do concurrency very well.

Collecting F3M Merging Metrics

Merging metrics are collected before function encodings since not all functions touched on IR2Vec would be merged in F3M, decreasing the amount of functions' vector representations to collect.

F3M provides a reporting feature that outputs the computed metrics for every function pair's merging performance. To collect merging metrics, F3M's implementation within the LLVM codebase is repurposed to use SQLite's C++ interface, piping the pairwise metrics directly into the database instead of the terminal or log files. This approach simplifies the implementation and decreases data collection time by eliminating any post-processing of log files. After these changes, LLVM's optimiser is run on the bitcode files, invoking the F3M pass with the reporting flag to collect all pairwise data.

After running the scripts from this step, all fields in the new database should be populated except for the encoding field, which will be populated in the next step.

Collecting IR2Vec Function Encodings

Function encodings are collected separately because each function is encoded only once while merging metrics are computed for every function pair.

IR2Vec's binary was used to generate function-level embeddings from the bitcode files with flow-aware embeddings to better capture the relational aspects between the components in the file. A compatibility issue was identified with IR2Vec. The function names used by IR2Vec and LLVM are different. LLVM's function retrieval returns mangled function names, while IR2Vec produces demangled names. Consequently, IR2Vec's source code was modified and rebuilt to conform to LLVM's naming convention.

The encodings are output and stored in text files, a medium provided by IR2Vec. An assistive script is then made to load the encodings from the text files into the database as a serialised binary object. After connecting to the database, the script retrieves the corresponding database entry after accessing it and updates it with the embedding.

Merging Data

Since multiple scripts were run concurrently while writing to different databases, a script was created to merge all data into a single, centralised database automatically. The script accepts a variable number of arguments specifying the input databases to combine and an argument for the new database's name. It works by initialising a new database using the specified name, connecting it to each existing database, and then copying over the information.

3.3 Model Development

To develop machine learning models, the data from the previous step are pre-processed and partitioned to form a representative dataset. Then, two ML model architectures were designed, and a framework was established to support efficient training, testing, fine-tuning, and evaluation of these models, streamlining experimentation and providing a reliable structure for iterative improvements.

3.3.1 Data Pre-Processing

Data Imbalance

A major challenge encountered with the dataset was the overwhelming number of function pairs with an alignment score of 0. In total, there were ***2.2 billion*** function pairs collected, of which ***1.67 billion*** samples had an alignment score of 0 (zero-samples), ***1.7 million*** had an alignment score of 1 (one-samples) and ***570 million*** has an alignment score between 0 and 1 (non-zero-non-one samples). If this imbalance is not adequately addressed, the model would likely learn to predict 0 for most situations to achieve a superficially low error but produce unhelpful predictions.

This imbalance is mitigated in two ways. First, a large portion of the zero samples is discarded so that the remaining zero samples equal the combined total of one and non-zero-non-one samples.

$$Zero\ Samples = One\ Samples + Non\_Zero\_Non\_One\ Samples$$

This balancing strategy reduces the dominance of zero samples and decreases overall training time by lowering the total number of training examples. Secondly, during model training, each remaining zero-alignment sample is assigned a weight of ***0.001***. This weighting means that every non-zero alignment sample contributes 1,000 times as much to the loss function as a zero-alignment sample, thus diminishing the influence of the overly abundant zero samples. This way, the model is incentivised to learn accurate predictions for non-zero alignment scores.

Data Split

After collecting and processing the data, we end up with ***1.1 billion*** function pairs. The order of data is then randomised to make it diverse when encountered by the machine learning model, preventing bias from improving generalisation and stopping the model from learning misleading patterns based on data sequence. After which, the dataset is split into three smaller SQLite datasets, the training, validation and testing datasets, each making up 70%, 10% and 20% of the pre-processed dataset, respectively. The model uses the training dataset to train itself. In contrast, the model uses the validation set to tune its hyperparameters (discussed in section 3.3.3). The test set is then used to test the model's performance on unseen data.

Pre-splitting the dataset accelerates training by eliminating the need to determine the data split during runtime, an extremely time-consuming process when working with a very large dataset. Furthermore, maintaining a permanent split throughout the model development stage reduces uncertainty in performance evaluations, as the deterministic nature of the split ensures consistent results. Consequently, any changes in metrics could be mainly attributed to the model's performance.

3.3.2 ML Model Design

The machine learning model developed will take in two functions' embeddings as inputs and produce an alignment score as the output.

Two model architectures were designed to predict the alignment score. The first is a Siamese model that uses dot product to quantify the similarity between the function embeddings. The second model features an attention mechanism to prioritise different elements of the inputs when making its predictions. Both models share the same goal with different approaches, the former modifies the input to fit the purpose better while the latter picks the most important information from the input to make its prediction. Comparing these methods allows us to evaluate which is more effective at alignment-score prediction.

Dot Product Siamese Model

This is an implementation of the Siamese network discussed in section 2.4.4, which uses a shared encoder to process two function encodings and then compare them using a dot product similarity. Figure 3.3 visually shows the architecture design of the dot product Siamese model.

Shared Encoder The shared encoder projects IR2Vec's embeddings into a suitable feature space for comparison using the dot product. This works by clustering functions that merge well with each other in the feature space so the dot product can produce higher scores for embeddings that are close.

The shared encoder first expands IR2Vec's 300-dimensional vector in 512 dimensions to try and capture any more semantics lost due to the limited amount of elements,

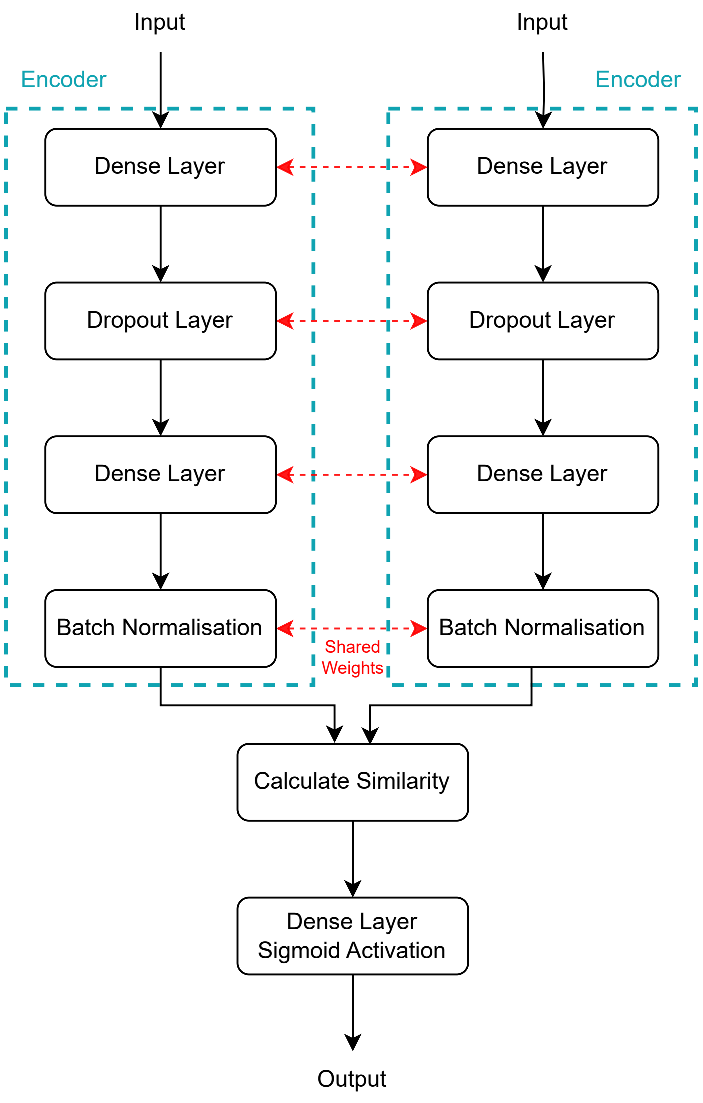


Figure 3.3: Visual representation of the Dot Product Siamese Model. The encoders for both models share the same weights to map both functions' features onto the same function space.

followed by a dropout layer with a dropout rate of 16.6% to prevent overfitting. Then, a dense layer compresses the new 512 dimensions into 128 to abstract and retain the most relevant information for the comparison task.

After this, batch normalisation is applied to standardise the layer outputs. This also reduces the need for careful initialisation and allows higher learning rates, effectively requiring one less hyperparameter to tune than a regular normalisation layer.

Score Generation Once both function embeddings have been processed through the shared encoder, the similarity between the embeddings is measured to capture how aligned the two functions are in the embedding space. Two similarity metrics, the dot product and cosine distance, are evaluated against each other to identify the most effective metric for this task. This similarity value is then passed through a dense layer of 1 unit with a sigmoid activation function to map the dot product to the [0,1] range, corresponding to the alignment score range.

Multi-headed Self-Attention Model

This is an implementation of a model which takes advantage of the attention mechanism discussed in section 2.4.3, visually shown by figure 3.4.

The model first processes each IR2Vec embedding through a shared dense layer with 512 units and ReLU activation, expanding the representation to capture richer semantic information. Unlike the Siamese model, these expanded representations are then reshaped and concatenated, allowing the model to treat the function pair as a unified sequence.

The core of this architecture is the attention block, which implements multi-head self-attention with four attention heads, each with a dimension of 64. This mechanism allows each function representation to attend to the other, capturing complex relationships between their features.

The self-attention output passes through dropout (30%) to prevent overfitting and uses skip connections and layer normalisation to stabilise training. The subsequent feed-forward network with ReLU activation further processes these attention-weighted representations through a dimension of 64 before returning to the original dimension. After the transformer encoding, the model applies global average pooling to aggregate the sequence information into a single vector representation. This condensed representation is passed through a final dense layer with sigmoid activation to produce an alignment score in the $[0,1]$ range.

By leveraging attention mechanisms, this model can potentially capture more intricate relationships between function pairs, learning which aspects of functions are most relevant for aligning functions.

3.3.3 Hyperparameter Tuning

Due to the sheer volume of training data available, tuning the hyperparameters using the entire training dataset would be computationally prohibitive. Therefore, the hyperparameter tuning stage uses a subset of ***300 million*** training samples. Once the optimal hyperparameters are identified through this procedure, the model will be retrained on the full dataset comprising 1.1 billion training samples using these optimal settings.

For the hyperparameter optimisation, ***Optuna*** is employed. Optuna is an efficient, automatic hyperparameter optimisation framework that utilises Bayesian optimisation methods to navigate the hyperparameter space. Each evaluation of a particular set of hyperparameters is referred to as a *trial*. During each trial, the model is trained on the subset of data, and its performance is measured on the validation set, specifically the mean squared error. The results from numerous trials guide the search process, balancing the exploration of new configurations and exploitation of known good regions

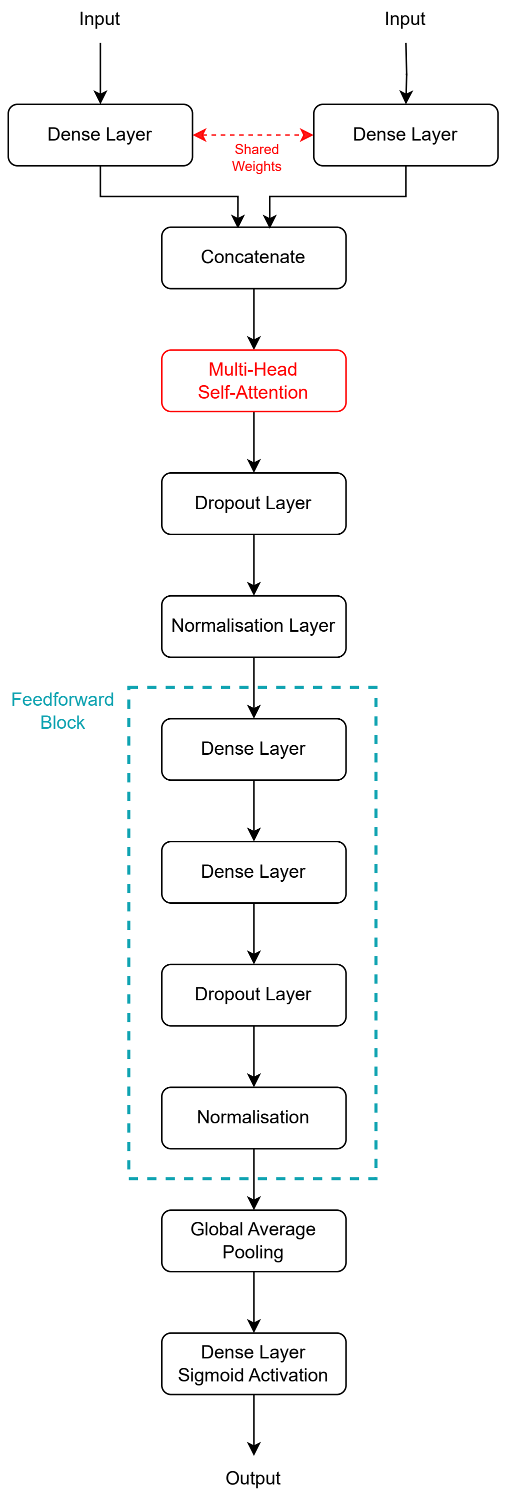


Figure 3.4: Visual representation of the Multi-Headed Self-Attention Model. The model uses the attention mechanism to allow each element to attend to another.

in the hyperparameter space, ultimately converging towards a model with the lowest mean squared error with the optimal settings.

3.3.4 Framework/Pipeline for Model Development

This section discusses the framework design and explains key design decisions.

Serialising Data

Initially, the models loaded data using Tensorflow's dataset interface, which acts as a lazy iterator for querying the SQLite dataset. Tensorflow's SQL dataset interface could not be used because the encodings, which were stored as BLOB objects, required depickling before being loaded into the model.

Direct querying of the SQLite dataset proved extremely slow. Therefore, a script to serialise the data into ***NumPy*** arrays was created. This approach significantly reduced the required processing operations, resulting in a speedup of approximately ***60%*** in loading and processing times compared to SQL querying.

The serialisation utilises two two-dimensional *NumPy* arrays: one to store function encodings and another to store alignment scores. Each unique tuple (BenchmarkID, FunctionID) is assigned a new unique function ID for the encodings array. These function IDs are stored with their respective encodings occupying the remaining columns. The second array stores alignment scores of function pairs using three columns: the first and second function's unique ID and the pair's alignment score.

The *NumPy* arrays are saved to binary files for direct loading when needed, speeding up the process compared to loading and parsing data from a text file. While the number of functions is manageable, it pales in comparison to the number of function pairs. To prevent overloading the system's memory when loading the pairwise metrics due to its volume, the array is split into multiple binary files with a specified number of samples (chunk size) in each file so that each file remains small and manageable for loading.

Selecting an appropriate chunk size during serialisation is crucial, as excessively large values can cause system crashes or deadlocks. During the serialisation process, the script verifies that at least 20% of system memory is available before retrieving the next data batch. If insufficient memory is available, the script sleeps until adequate memory becomes available. A potential deadlock can occur if the script holds data in memory, waiting to meet the size requirement, but lacks sufficient memory to retrieve the additional data needed to offload the current data into storage, freeing up memory.

Loading Data

This process utilises the TensorFlow dataset interface, where the dataloader first loads all function encodings into memory and then processes each chunk of pairwise metrics by replacing unique function IDs with their corresponding encodings.

This loading process was observed to be time-intensive. To address this performance bottleneck, the ***concurrent*** Python library was used to load the next data chunk in a separate thread while the model trains on the currently available data. These loaded chunks are placed in a queue, which the dataloader lazily iterates through, significantly reducing the time the model spends waiting for data preparation between chunks and accelerating the training process.

The data loader is configured to process only one additional file concurrently, as processing more files simultaneously provides no performance benefit and consumes excessive system memory. This is particularly important since each fully expanded input contains 601 32-bit float values, around 2KB (A.2).

To prevent system crashes due to memory exhaustion, the loader verifies that at least 20% of system memory is available before loading and expanding each new data chunk. This memory management strategy ensures stable operation throughout the training cycle.

Running the Model

To streamline the model development process, a centralised script was developed that efficiently handles training and hyperparameter tuning across different model architectures. The main script accepts various command-line arguments to specify the model type, the operational mode (training or hyperparameter tuning), the location to save the trained models and, when in training mode, the specific model parameters.

For this modular approach to function effectively, each new model must implement two standard interface functions: $get\_model()$ and $HyperParameterTraining()$. The $get\_model()$ function provides the main script access to the model, while $HyperParameterTraining()$ encapsulates any model-specific hyperparameter optimisation requirements. This standardised interface eliminates redundant code and significantly simplifies working with different model designs.

A ***Jupyter Notebook*** was created for model evaluation and result visualisation. This notebook tests trained models against the test dataset, calculates the mean squared error, and generates comparative plots of predicted versus true alignment scores to demonstrate model performance visually.

3.4 System Integration

This section describes how IR2Vec and the trained ML models were integrated into the F3M's codebase to leverage the ML models for function merging decisions.

3.4.1 Integrating Tensorflow

The initial integration strategy involved using ***PyBind*** to load the trained TensorFlow models into the compiler. This approach was selected because one variant of the Siamese Model had been developed using the L1 distance metric instead of the dot product. Since this implementation required a custom layer not natively available in TensorFlow, PyBind appeared advantageous for running Python code, allowing custom layers to be used.

However, PyBind consistently crashed when attempting to load TensorFlow models. Given these limitations, *TensorFlow's C++ API* was adopted as an alternative solution for loading the trained models. This decision required abandoning the L1 distance Siamese model due to the engineering work required to support its custom layer in the new integration approach.

3.4.2 Integrating IR2Vec

Next, since IR2Vec was developed in C++, the initial approach involved incorporating its header files and source code from the official *GitHub* repository. However, while testing the project, errors occurred whenever the compiler attempted to request embeddings from IR2Vec. This compatibility issue likely stemmed from version differences between the LLVM versions used by each project. IR2Vec was based on LLVM 18.1.8 at the time of this project, while F3M used version 18.0.0, necessitating an alternative solution.

The adopted workaround involved pre-generating function encodings using IR2Vec's stand-alone executable for the benchmarks and storing them in text files. During compilation, the system would then parse these files and load the encodings into a map for quick encoding lookup of each function.

This integration approach imposed a significant limitation: newly generated merged functions could not be considered candidates for further merging, as it was impossible to obtain embeddings for these new functions during compile time. This constraint restricted the optimisation potential to a single pass of function merging.

3.4.3 How it all fits together

The function merging process begins when all candidates are placed into a priority queue. As each candidate is processed, it is removed from the queue for assessment. The queue prioritises functions with larger fingerprint sizes, giving them precedence over smaller ones. During assessment, the system predicts the alignment score between the current function and all other candidates in the queue. Merging is only attempted with the function that yields the highest predicted alignment score.

Thresholding Alignment Score Even when the highest-scoring function pair is identified, merging only proceeds if the alignment score exceeds a predetermined threshold value. This threshold mechanism prevents wasting compilation resources on unprofitable merges when a function may be unsuitable for merging with other functions in the queue. Multiple thresholding values were tested during evaluation, including 0.4, 0.5, and 0.6, to determine the optimal balance between merge opportunities and compilation efficiency.

3.5 Artifactory and Set up Script

3.5.1 Repository and Accessibility

All code developed for this project is publicly available as artifact in a GitHub repository, linked in appendix A.3. This repository contains the complete codebase, including models, data processing scripts, integration components, and evaluation tools. The repository is structured to facilitate easy navigation and understanding of the project's components, with clear documentation for each module.

3.5.2 Automated Setup Script

To enhance accessibility and reproducibility, a comprehensive setup script has been developed that automates the installation and configuration process. The script applies custom configurations to ensure compatibility between components and prepares the system for immediate experimentation. This automation significantly reduces the technical barriers to reproducing the research.

A detailed README file is included in the repository root, providing comprehensive guidance through the project setup process. The documentation clearly outlines the required dependencies and tested environments to ensure compatibility. It includes straightforward installation instructions that walk users through the cloning process and execution of the setup script.

The setup script intentionally excludes the complete installation of IR2Vec, instead only cloning the repository and applying necessary patches. This decision was made because IR2Vec requires a local build of LLVM as a prerequisite, requiring extensive disk space and compilation time.

Chapter 4

Evaluation

In this section, this project will be evaluated according to three criteria introduced in section 1.2. First, the trained models' performance on the test set is analysed. Second, all merge attempts are examined for their profitability to evaluate the quality of the merge attempts. Third, the size of the code is measured to assess the impact of merging on the code footprint.

4.1 Experimental Set Up

The following experiments were run on an Intel Xeon Gold 6338 CPU with 128 threads and two Nvidia A10 GPUs running AlmaLinux. The results are reported on the SPEC CPU 2006 and SPEC CPU 2017, presenting individual benchmark outcomes. Since F3M is non-deterministic, the results are averaged over three runs.

4.2 ML Model Prediction Accuracy

To assess the trained deep learning models' prediction accuracy, the models are evaluated against a test set of *228 million* unseen samples. We quantify the performance of the models using TensorFlow's built-in evaluation feature, which computes the mean squared error (MSE) between each prediction and its true value. MSE is a standard metric for evaluating and comparing regression models in machine learning. Furthermore, a heat map is generated to visualise the model's performance, plotting the model's predicted alignment score against the true label to visualise the model's performance.

Given the data imbalance (3.3.1), two MSE values are calculated, one using the whole test set and one using only non-zero samples from the test set. This approach addresses bias compensation, the imbalanced dataset might inherently bias the model toward predicting lower alignment scores, and the separate MSE helps quantify whether the model successfully overcomes this training bias. Additionally, this helps us quantify the model's performance for functions where there are similarities between them, as this is more important for our purposes.

4.2.1 Dot Product or Cosine Similarity

During the model selection process, two variants of the Siamese neural network were designed and evaluated, one using the cosine similarity and another using the dot product as the similarity metric. A subset of approximately *80 million* training samples was used for initial model evaluation to efficiently determine the most promising architecture before proceeding to hyperparameter tuning and training on the full dataset.

Scatter plots showing predicted alignment scores against actual alignment scores were generated to compare the models' predictive capabilities. These visualisations excluded data points with a true alignment score of 0 to focus specifically on the models' ability to predict non-zero alignment values. Both models were trained with identical hyperparameters for six epochs each, the only difference being the similarity metric used.

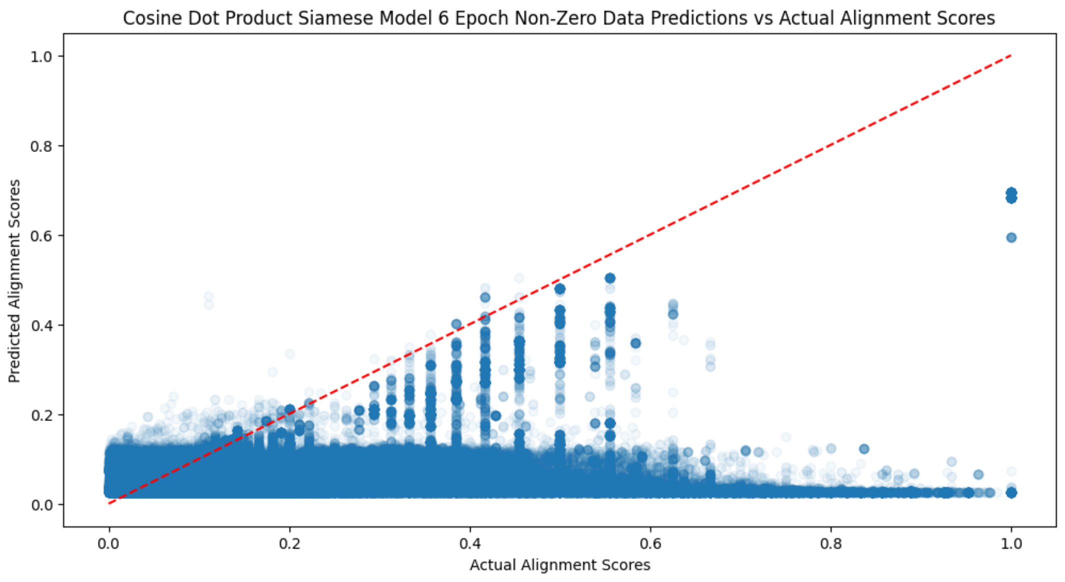


Figure 4.1: Scatter Plot of Cosine Similarity Siamese Model's Predictions vs. True Labels using 80 million training samples. The diagonal line represents perfect agreement. Scatter plots effectively reveal patterns in smaller datasets; however, they lose clarity as datasets grow larger. In the subsequent sections, the volume of data is significantly larger, so heat maps were used to capture density and trends better.

Comparing figures 4.1 and 4.2, it is observed that the dot product model is able to demonstrate superior generalisation and prediction compared to the cosine model. The improved results achieved by the dot product model relative to the cosine model may be attributed to the inability of cosine similarity to capture function size information. The alignment score inherently depends on both the similarity of instructions and the total number of instructions in each function, making function size a key determinant (2.2.2). This limitation is particularly relevant when working with IR2Vec embeddings, which construct function representations through summation of the basic blocks and instructions, naturally encoding size information in vector magnitudes. The

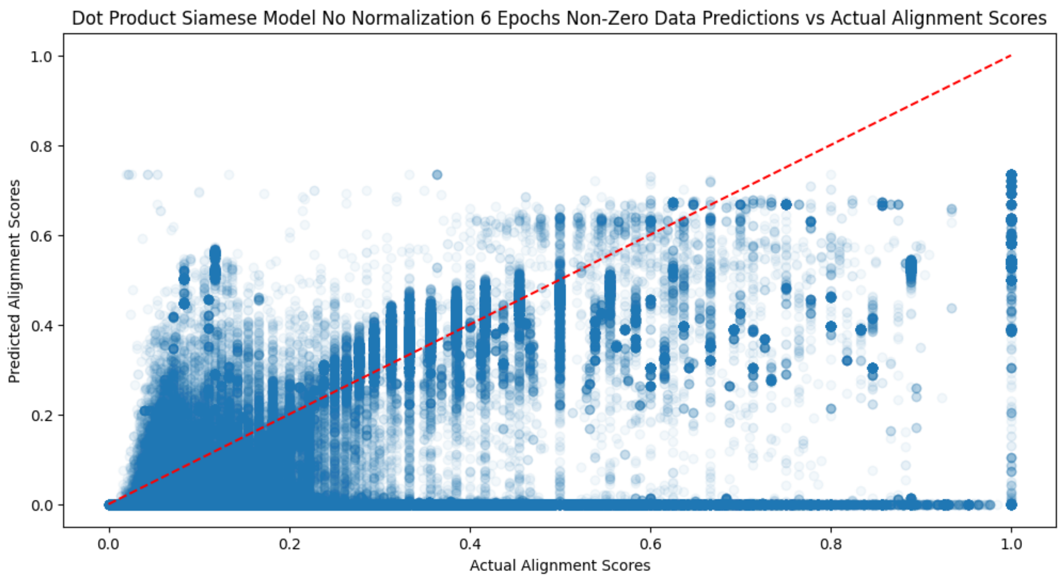


Figure 4.2: Scatter Plot of Dot Product Siamese Model's Predictions vs. True Labels using 80 million training samples. The diagonal line represents perfect agreement. Scatter plots effectively reveal patterns in smaller datasets; however, they lose clarity as datasets grow larger. In the subsequent sections, the volume of data is significantly larger, so heat maps were used to capture density and trends better.

normalisation in cosine similarity causes the model to treat two functions with proportionally similar structures identically, regardless of their sizes, despite having potentially very low alignment scores in practice. Based on these findings, the dot product was selected as the preferred metric for the final implementation of the Siamese model.

4.2.2 Dot Product Siamese Model Results

The fully trained and tuned dot product Siamese model demonstrated strong performance, achieving an MSE score of **0.0319** on the complete dataset and a **0.0043** on the non-zero test set.

Figure 4.3 shows that the model achieves good prediction accuracy for alignment scores up to *0.5*, with most predictions falling near the diagonal line representing perfect agreement. However, beyond this threshold, the model's predictions form a visible horizontal band plateauing at around *0.7* in the predicted score range, making it less reliable for identifying highly aligned functions. This creates a ceiling effect where functions with true alignment scores between *0.7-1.0* are consistently underestimated.

This limitation could be attributed to the imbalanced training data distribution, as evident from the colour intensity in the lower regions of the heat map, where the colour intensity indicates a significantly higher concentration of examples with lower alignment scores. Consequently, the model optimises its performance by learning to predict lower scores more frequently, minimising the overall loss function but compromising accuracy for high-alignment cases.

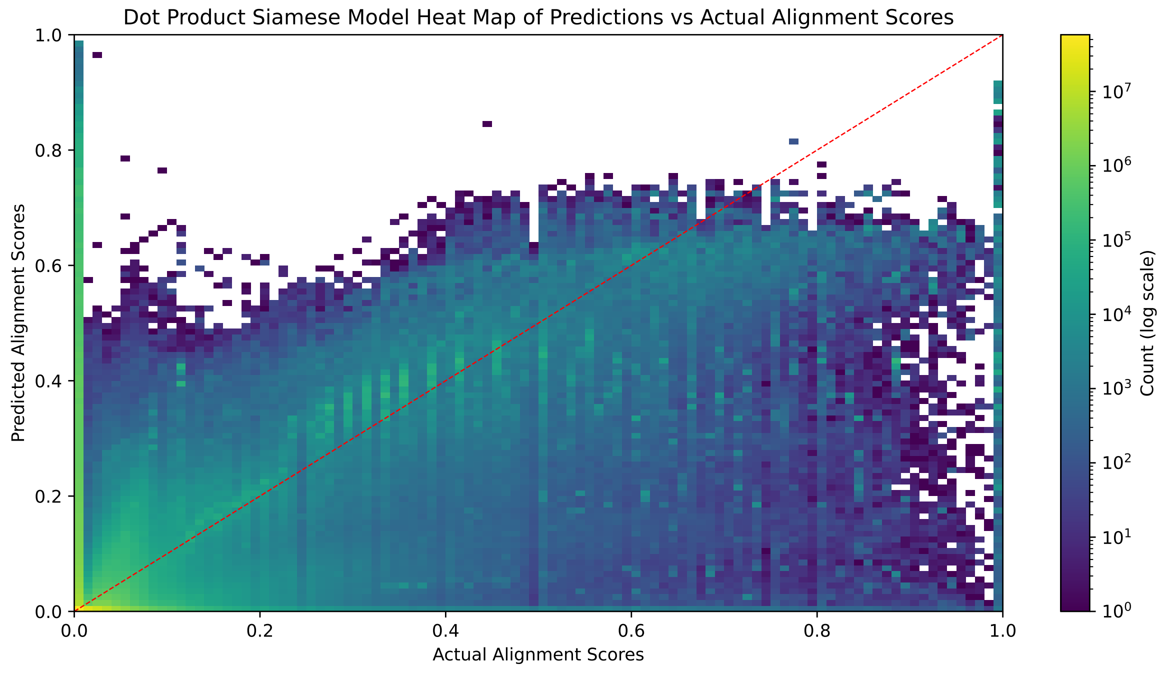


Figure 4.3: Frequency Heat-map of Dot Product Siamese Model's Predictions vs. True Labels. The colour intensity (log-scaled count) represents the frequency in each prediction–actual score bin, and the diagonal line represents perfect agreement.

4.2.3 Multi-Headed Self-Attention Model Results

The fully trained and tuned multi-headed self-attention model demonstrated exceptional performance, achieving an MSE score of **0.00738** across the entire testing dataset and **0.00094** for the non-zero testing dataset, orders of magnitude lower than the Siamese model's MSE.

Figure 4.4 reveals that this model reliably predicts true alignment scores across the full spectrum of values, with most predictions closely following the diagonal line that represents perfect agreement. The heat map shows a more consistent prediction pattern along the diagonal, particularly in the 0.7-1.0 range, where the Siamese model struggled. While significantly outperforming the dot product Siamese model, this attention-based architecture still exhibits a subtle bias toward under-prediction rather than over-prediction when errors occur. However, this tendency is considerably less pronounced than in the previous model, allowing for more accurate identification of highly aligned function pairs.

4.2.4 Evaluation

Analysis of both models reveals a common pattern in figures 4.3 and 4.4, where samples with a true score of 0 are frequently misclassified. This observation, however, should be contextualised by the significant class imbalance in the dataset, with zero-alignment samples vastly outnumbering others, as indicated by the bright yellow/green

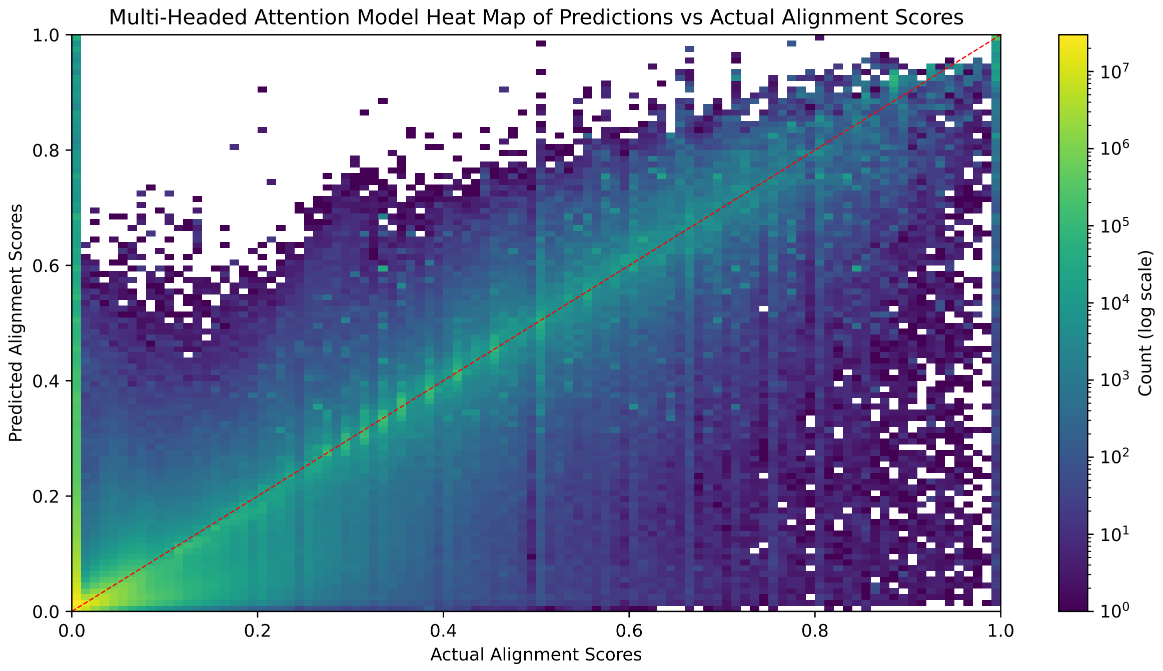


Figure 4.4: Frequency Heat-map of Multi-Headed Self-Attention Model's Predictions vs. True Labels. The colour intensity (log-scaled count) represents the frequency in each prediction–actual score bin, and the diagonal line represents perfect agreement.

regions at the origin of both heat maps. Notably, for both models, prediction errors decrease as the predicted values move away from 0, indicating robust capabilities even with imbalanced training data.

When directly comparing performance, the self-attention model demonstrates substantially superior prediction accuracy across the entire alignment score range, particularly for the higher alignment scores where the Siamese model's plateauing effect limits its utility. The visual difference between the two heat maps is striking. The attention model shows a much more defined diagonal pattern throughout the entire range, especially above *0.7*. The practical implication of implementing the Siamese model in a production environment may result in missed opportunities for merging highly-aligned function pairs due to systematic under-prediction, potentially reducing the system's overall effectiveness. Conversely, the attention model's more balanced error distribution makes it a more reliable choice for accurately identifying candidates for function merging across the full spectrum of alignment scores.

The comparative analysis reveals that the attention architecture is more forgiving towards data imbalance, maintaining high performance even with minimal preprocessing techniques. On the other hand, the Siamese architecture exhibits clear limitations when trained on skewed distributions, suggesting that it would benefit from more sophisticated pre-processing approaches. This fundamental difference in how each architecture handles class imbalance represents an important consideration for deployment scenarios where balanced training data cannot be guaranteed.

4.3 Quality of Merging Predictions

Next, we evaluate how well the predicted alignment scores serve as indicators for profitable function merging compared to F3M. Although the ML models may accurately estimate alignment values, those predictions may not translate into correct merge decisions. To assess and compare the quality of the merging decisions between F3M and the ML approach, function merging was applied on *SPEC CPU 2006* and *SPEC CPU 2017* benchmarks, using the model to predict scores for previously seen function pairs.

The function selection process for the predictive approach mirrors F3M's search method for the optimal merge candidate for each function (2.2.2). Instead of using MinHash and LSH, it finds the optimal candidate by selecting the function with the highest alignment score. If this score exceeds a pre-determined threshold, 0.4, 0.5 and 0.6 are tested, the two functions are considered for merging. This thresholding strategy aims to reduce the compile time by eliminating low-potential merging attempts that would likely prove unprofitable. The merging process yields two outcomes, whether it is possible to merge the functions (**validity**) and, if valid, whether the merged function is predicted to be **profitable** in terms of binary code size reduction by using LLVM's internal cost model to estimate the function sizes.

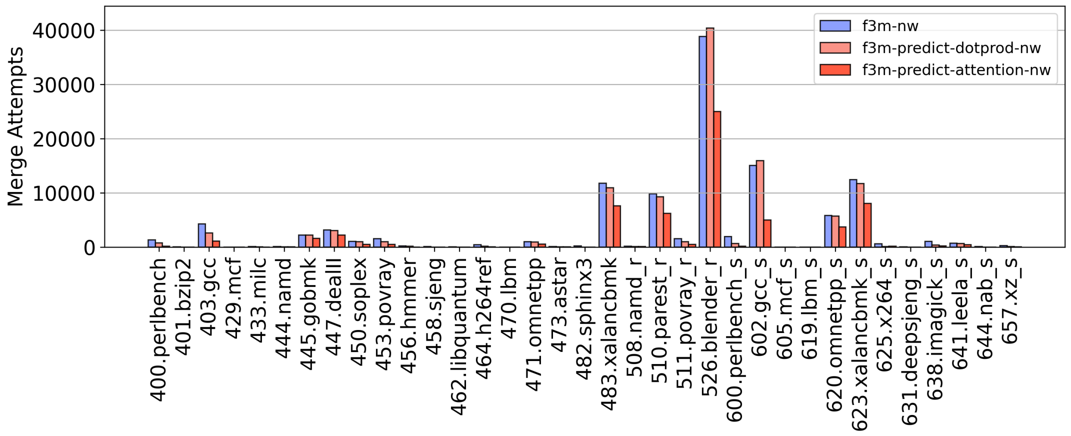
This evaluation employs two metrics in combination to quantify the quality of the predictions, the number of total merges made and the number of profitable merges, to calculate the percentage of profitable merge attempts (A.1). These metrics will be plotted for each benchmark to visualise their merging quality.

4.3.1 Number of Attempted Merges

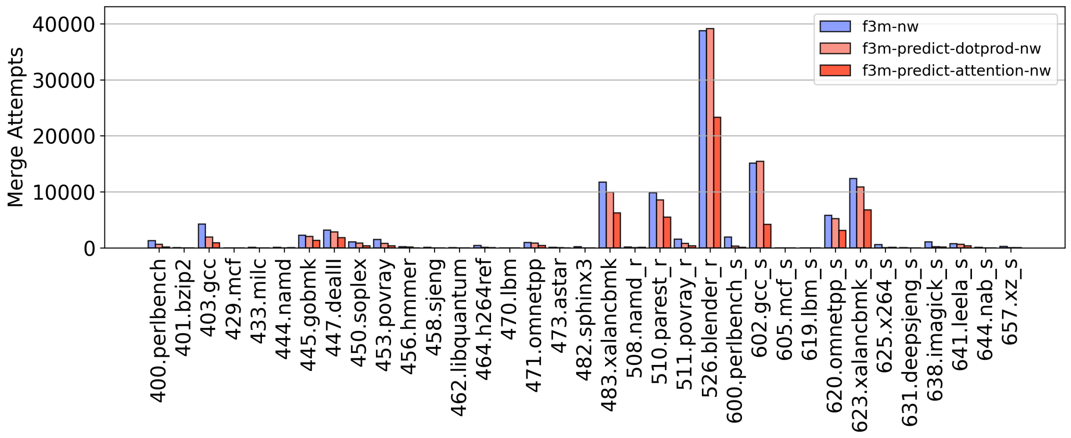
Figure 4.5 collectively shows the number of attempted function merges for F3M, dot product model prediction approach and the attention model prediction approach across all prediction threshold values.

The benchmark *526.blender_r* exhibits the most merging attempts, more than double that of the next highest benchmark, while *429.mcf* and *605.mcf_s* have the least attempted merges. This disparity indicates that blender contains many function candidates for function merging operations, while mcf offers a limited number of options, restricting the potential for merging due to the lack of available pairs.

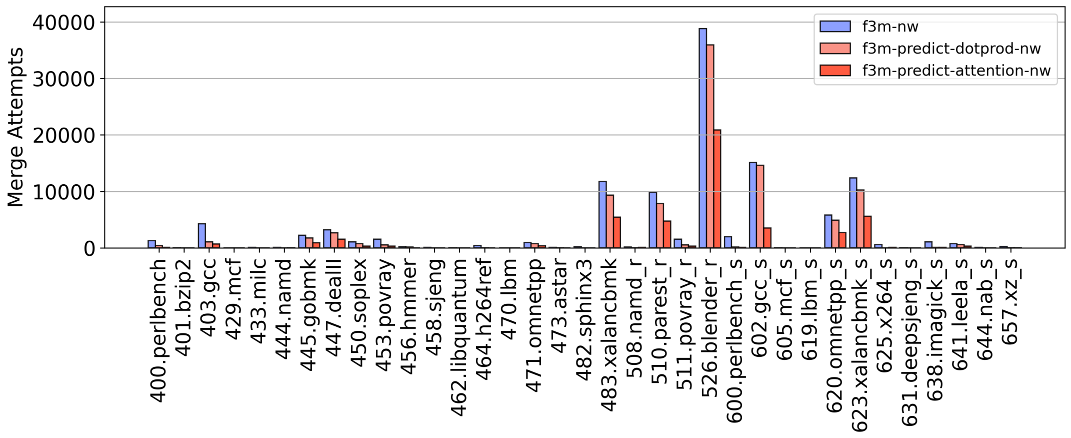
As the threshold increases, both predictive models demonstrate an inverse correlation with the number of merge attempts as expected. This effect is particularly pronounced in the dot product model, which initially attempts more merges than F3M in some cases at a threshold of 0.4, matches F3M generally at 0.5, and finally, fewer merges at a threshold of 0.6. In contrast, the attention model consistently predicts significantly fewer suitable functions to merge than the other two implementations across the whole benchmark suite, with some cases such as *602.gcc_s* showing more than 50% fewer attempts.



(a) Number of Attempted Merges (0.4 Threshold)



(b) Number of Attempted Merges (0.5 Threshold)



(c) Number of Attempted Merges (0.6 Threshold)

Figure 4.5: Number of Attempted Merges for each threshold.

4.3.2 Profitability of Attempted Merges

Finally, figure 4.6 graphs the percentage of merges that produced a profitable merged function across all 35 benchmarks at three similarity thresholds. Profitability is determined when the estimated size of the newly merged function is smaller than the sum of the estimated sizes of the original function pair.

Overall, the dot-product approach struggles to identify merge-able pairs regardless of threshold. It rarely breaks the 2.5% mark, achieving its best result of 29% on the *433.milc* benchmark at the 0.6 threshold. On the other hand, the attention-based approach matches or exceeds F3M on 18 out of 35 benchmarks (just over half), peaking at a 47.5% profitable-merge rate on *638.imagick_s* when thresholded at 0.6.

4.3.3 Evaluation

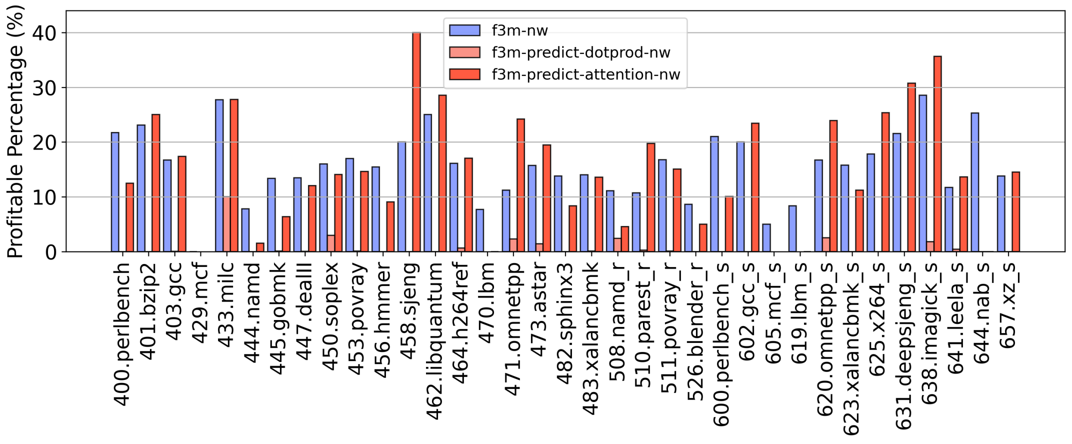
The Dot Product Model demonstrates significant limitations as a heuristic for function merging. Despite proposing similar merges to F3M, its precision remains unacceptably low, topping out at 29% valid merges on the *433.milc* benchmark at a 0.6 threshold. This indicates that over 70% of its attempted merges are unprofitable and cannot identify profitable function pairs, representing substantial wasted computational effort during compilation, rendering it an unreliable indicator of merging performance.

The attention model presents a more conservative approach than F3M, generating significantly fewer merge attempts than F3M, sometimes less than half as many. This reduction raises two possible interpretations, either the model fails to identify viable merging opportunities that F3M captures or achieves superior global optimisation by prioritising the most beneficial merges first, leaving fewer similar functions available for subsequence merges. When profitability is considered, the attention model maintains consistent performance across all tested thresholds, showing promising prediction quality.

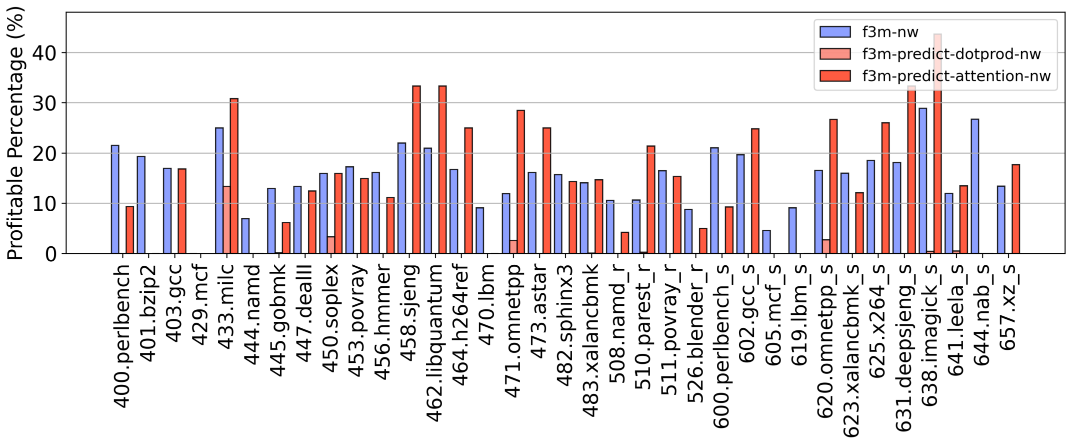
The evidence indicates that the Attention Model substantially outperforms the Dot Product Model as a function merging heuristic. Despite its high proposal volume, the Dot Product approach suffers from a poor, unprofitable merge rate. In contrast, the Attention Model delivers noticeably better profitability across all thresholds. Further evaluation is needed to determine whether the Attention Model's reduced attempt count represents optimality or overlooks merges.

4.4 Code Size Reduction

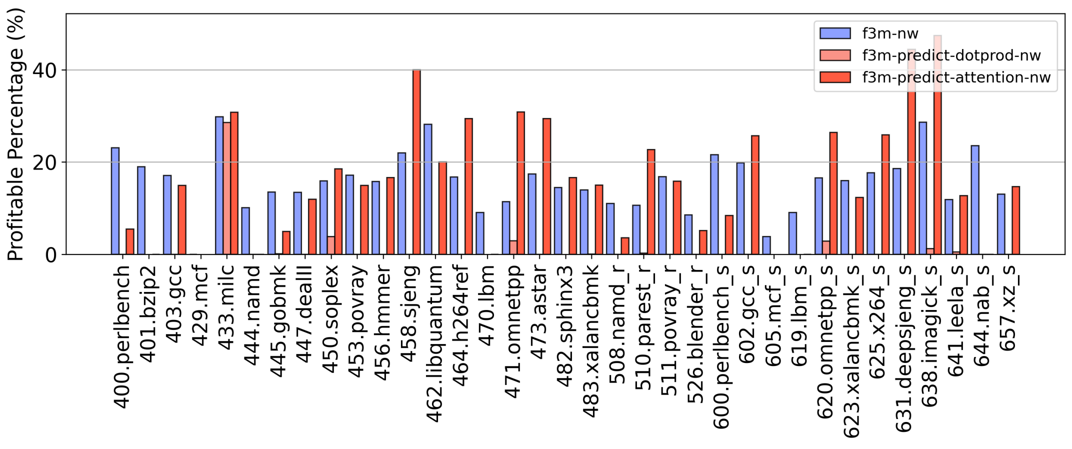
Lastly, we analyse the effect the three approaches, F3M, Dot Product Predictions and Attention Predictions, have on code size reduction compared to LLVM's default compiled binary. There are two metrics this evaluation focuses on, the size of the compiled-code segment of the binary file (also known as *.text*) and the size of the binary file (*binary size*). The compiled-code section of an executable contains only the actual machine code instructions, holding the code that will be executed at runtime. The binary



(a) Profitable Merge Attempts (0.4 Threshold)



(b) Profitable Merge Attempts (0.5 Threshold)



(c) Profitable Merge Attempts (0.6 Threshold)

Figure 4.6: Percentage of merge attempts that are profitable for each threshold. This graph illustrates the proportion of merge attempts that result in smaller function size calculated using equation A.1.

size, on the other hand, refers to the total size of the compiled file. This includes all the machine code instructions, global variables, read-only data, and other metadata specific to the executable format. The compiled-code section size provides direct insight into the effectiveness of function merging optimisation, while the binary size allows us to understand the overall impact on the executable size.

4.4.1 Compiled-Code Segment Size Reduction

Figures 4.7 presents the compiled-code section size reduction achieved by the three function merging approaches (F3M, Dot Product Predictions, and Attention Predictions) across the benchmark suite compared to the baseline LLVM compilation. This metric is particularly relevant as it isolates the impact on actual executable code size, allowing us to evaluate each method's capability to identify and merge similar functions while excluding other factors that may affect the overall binary size.

F3M achieved an average compiled-code section size reduction of **2.9%** across SPEC CPU 2006 and SPEC CPU 2017 benchmarks, while our machine learning-based approaches demonstrated superior performance with reductions of **4.4%** and **4.2%** for the Dot Product and Attention models respectively. This represents a significant improvement of approximately 50% over the state-of-the-art F3M technique.

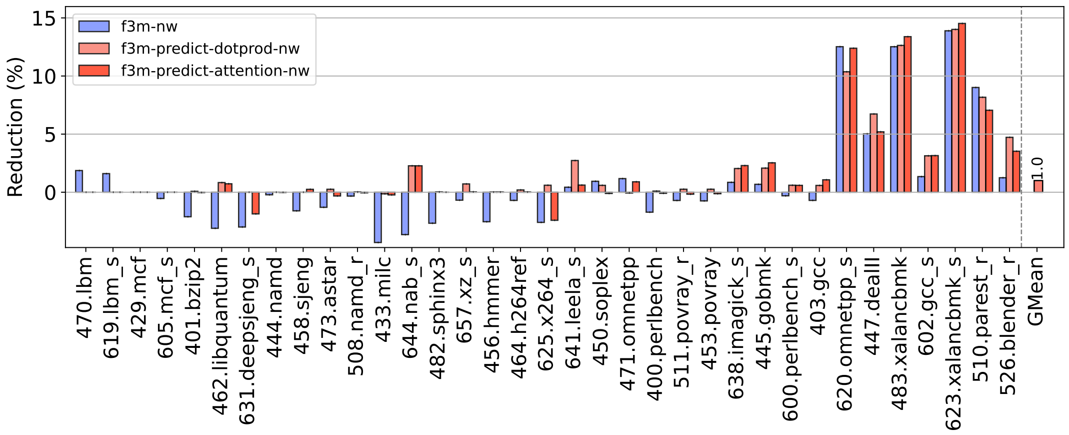
While examining individual benchmark performance, F3M consistently outperformed our predictive models on four benchmarks (450.soplex, 471.omnetpp, 620.omnetpp_s, and 510.parest_r) across all threshold configurations. However, our predictive approaches demonstrated greater consistency in reducing the compiled-code segment size, particularly on smaller benchmarks where F3M frequently increased the size rather than reduced it.

Regarding threshold sensitivity, the Dot Product model showed identical performance at thresholds of 0.5 and 0.6, with minimal difference when lowered to 0.4. The Attention model exhibited even more stable behaviour, with reduction percentages ranging narrowly from 4.24% to 4.29% across different thresholds. This shows that the threshold does not play a significant role in the compiled-code section's size at the granularity of thresholds we assessed.

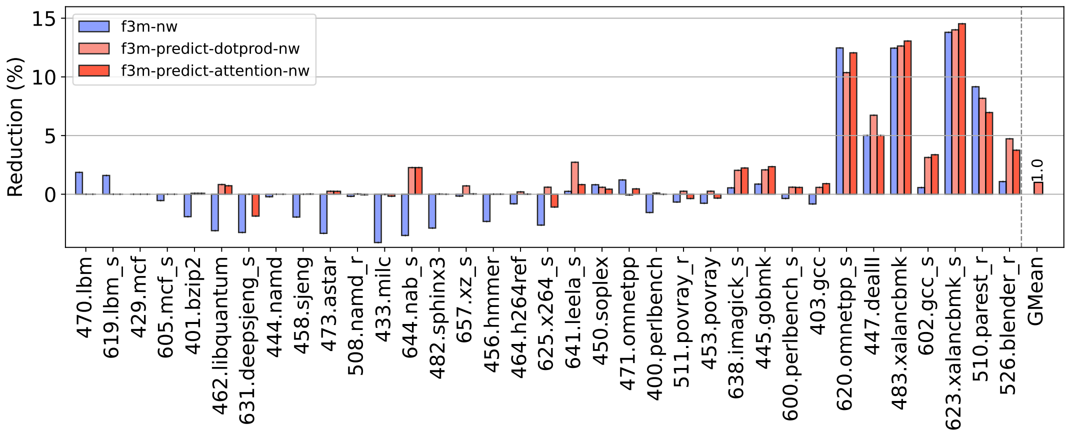
In summary, both the Dot Product and Attention model frequently surpassed F3M's capabilities, demonstrating that our machine learning approach successfully captures function similarities that traditional heuristic-based methods miss.

4.4.2 Binary Size Reduction

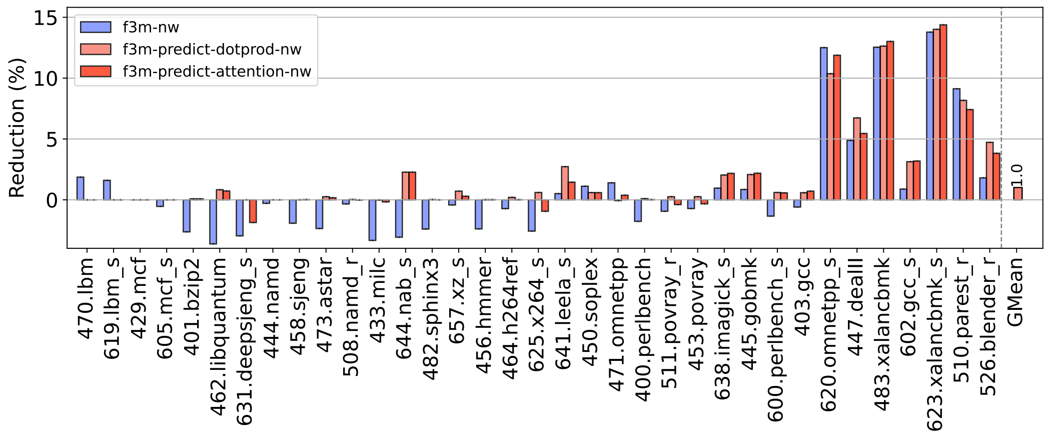
The binary size reduction provides a more comprehensive view of optimisation impact, capturing potential overheads in metadata, branch tables, and other sections that might be introduced during function merging. This metric is important for evaluating real-world deployment scenarios where total storage requirements matter. Figures 4.8 show the overall binary size reduction achieved by our three function merging approaches across SPEC2006 and SPEC2017 benchmarks compared to the baseline default LLVM compilation.



(**a**) Dot Text Size Reduction with **0.4** Threshold)

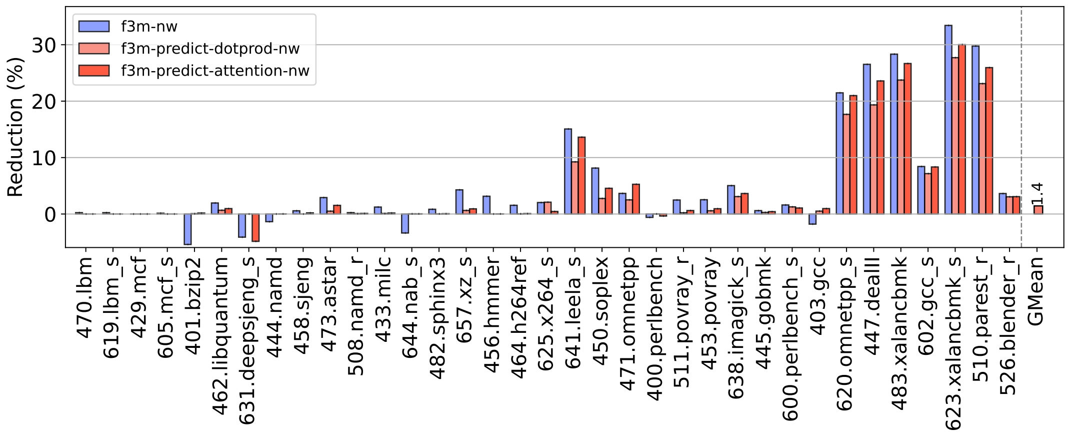


(**b**) Dot Text Size Reduction with **0.5** Threshold)

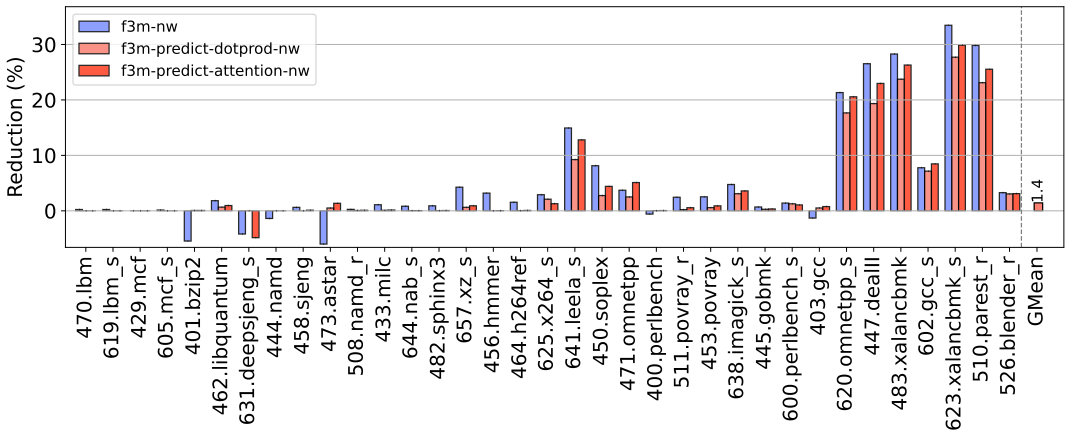


(**c**) Dot Text Size Reduction with **0.6** Threshold)

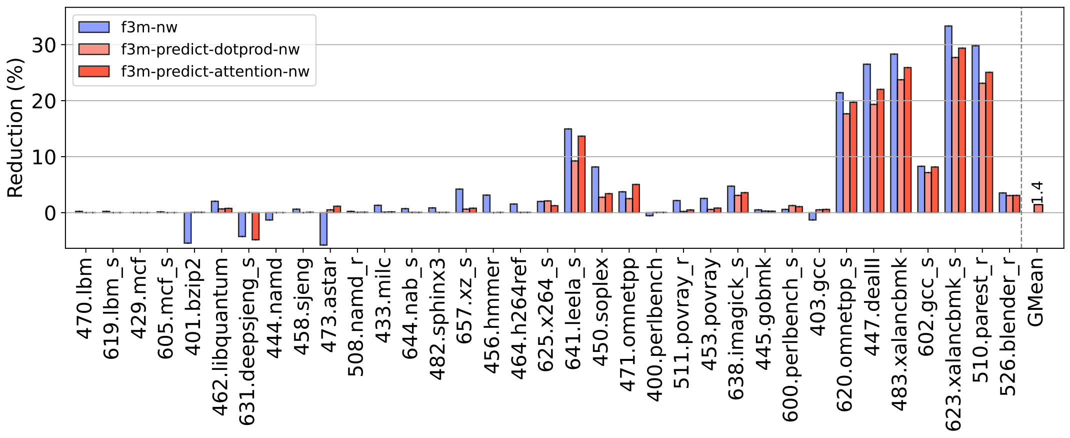
Figure 4.7: Dot Text Size Reductions Across Different Thresholds. The percentage of dot text size reduction is shown against the baseline, which is the default LLVM without any experimental function merging (Higher is better).



(a) Binary Size Reduction (0.4 Threshold)



(b) Binary Size Reduction (0.5 Threshold)



(c) Binary Size Reduction (0.6 Threshold)

Figure 4.8: Binary Size Reductions Across Different Thresholds. The percentage of binary size reduction is shown against the baseline, which is the default LLVM without any experimental function merging (Higher is better).

Unlike the compiled-code segment size results, figure 4.8 reveals the traditional F3M approach maintains an edge in overall binary optimisation, achieving a substantial **12.1%** reduction across our benchmark suite. Our machine learning techniques demonstrated respectable but lesser reductions of **9.9%** (Dot Product) and **11%** (Attention), suggesting F3M's heuristics may better address factors beyond code sections that influence total executable size. The Attention model was able to show promising results on three benchmarks (*471.omnetpp*, *403.gcc*, and *602.gcc_s*), where it outperformed F3M across all threshold configurations.

The predictability of our ML-based approaches stands as their most compelling advantage. The Dot Product model exhibited remarkable consistency, producing zero adverse outcomes across all benchmarks, a guarantee against unwanted size increases that F3M cannot provide. Similarly, the Attention model triggered negative reduction in just one case (*631.deepsjeng_s*), vastly outperforming F3M's unpredictable behaviour on smaller benchmarks. In comparison, F3M increased binary size on multiple smaller benchmarks, making our approaches more reliable for size-sensitive applications.

Threshold configuration analysis revealed minimal sensitivity across both models. The Attention implementation showed slightly better performance at lower thresholds (ranging from 10.7% to 11.1% reduction), but the practical difference remains negligible for most deployment considerations. The Dot Product model's results across all thresholds are almost identical, further emphasising this stability.

4.4.3 Evaluation

None of the implementations could make a breakthrough with both mcf benchmarks, regardless of threshold, likely because there are no possible merges. Despite this limitation, both predictive models showed promising results when examining how effectively machine learning can eliminate redundant code.

If we look at the compiled-code section size reduction, the predictive approach worked better than F3M, showing approximately 50% improvement. This metric best demonstrates the effects of function merging on code. It indicates superior performance compared to the previous state-of-the-art, representing a significant improvement for both predictive models. Surprisingly, the dot product approach performs well despite its poor merging prediction quality, functioning similarly to F3M by helping shortlist functions for the compiler to attempt merges.

While the predictive models work quite well, with their ability to decrease binary size, they fall slightly behind F3M, representing a 13% decrease in reduction compared to F3M, but do keep it in mind that this reduction was achieved without any handcrafted heuristic used to decide the function merges.

One reason for this discrepancy can be attributed to the data stored for exception handling. The exception handling frame (.eh_frame section) stores the call frame information (CFI), which describes how to restore the stack state at any point in program execution, necessary for stack unwinding during exception propagation. The exception handling frame header (.eh_frame_hdr section) is used for binary searching into the exception handling frame during runtime to quickly locate the appropriate frame description entry for a given program counter value, optimising exception handling.

When functions are merged, the CFI must maintain precise stack unwinding instructions for each possible execution path through the merged function. This requires additional CFI directives to track stack unwinding information across multiple execution paths that previously existed in separate functions. The merged function must maintain all exception-handling capabilities of the original functions while managing them within a single code body, leading to a substantial increase in exception metadata despite the reduction in code size.

The exception handling frame for the dot product and attention models are 13.6% and 8.5% larger than F3M's and 11.7% and 6.9% larger for the exception handling frame header. This shows an inverse relationship between the amount of compiled-code size and the exception-handling metadata. This demonstrates how function merging can lead to smaller compiled-code sections but larger overall binaries due to this metadata expansion.

If users are willing to take a slight risk when running function merging, it is better to use the attention model than dot product because the binary size will likely be smaller if the code size is reduced, though there is a slight chance that the size will increase slightly. On the other hand, dot product is safer, much less likely to generate a larger size for both the compiled-code section and binary size, but the reduction is less aggressive. Performance is also left on the table since the merged functions were not considered for merging again due to IR2Vec's versioning issues.

The compile time for the dot product and attention model approaches are 10 and 23 times longer than F3M. This is due to the implementation where the alignment score is predicted between a function and all other functions before merging with the highest aligned function. This process is costly and could benefit from a multi-tiered analysis process where earlier stages prune off non-profitable functions.

Chapter 5

Conclusion

This section concludes this project by summarising the work done and discussing the achievements, reflections and ideas for future works.

5.1 Summary

This project aimed to improve F3M's function-merging capabilities by building a robust data infrastructure, end-to-end ML pipeline, and seamless integration of alignment score predictions into the function selection process. Initially, a data-collection framework was designed and implemented that automatically gathers two complementary sources of information. First, F3M's reporting feature was enhanced to directly record each function merging pair and their alignment scores into our database. Second, IR2Vec was modified to generate and label function embeddings using the same mangled function names that F3M produces. These are stored in a database efficiently whose schema relates each function's metadata and vector representations to their merge attempts.

After obtaining the data for training, a centralised ML pipeline was developed in Python to accelerate model experimentation and deployment. The raw data undergoes pre-processing to address class imbalances, ensuring the model learns equally from well-aligned and badly-aligned pairs. A single entry-point script orchestrates training, hyperparameter tuning, and evaluation across multiple classifiers, providing a consistent interface for rapid iteration. The curated dataset is serialised into Numpy arrays to further speed up training cycles, allowing models to load data instantly without slow database queries.

The F3M codebase was extended with trained models to leverage alignment predictions for the function selection process. Due to versioning conflicts between IR2Vec and F3M's LLVM toolchain, embeddings are pre-computed and stored in text files. These embeddings, alongside exported TensorFlow models, are then loaded into the C++ merger via TensorFlow's C++ API. At runtime, F3M consults the predicted alignment scores to identify the highest-scoring function pairs to attempt function merging, replacing its original heuristics with data-driven decisions.

Finally, this project was evaluated across three dimensions: the ML model's accuracy on a held-out test dataset, the quality of merges produced by F3M's hand-crafted heuristic versus our dot-product siamese model-based heuristic and our attention-based heuristic, and the resulting code-size reduction under each strategy. This comprehensive analysis demonstrates that while our project fails to outperform F3M on the overall binary size reduction due to factors outside our control, integrating learned alignment predictions improves merge decisions and yields measurable gains in compiled-code segment shrinkage, showing the value of marrying compiler-level optimisations with machine learning.

5.2 Achievements

In line with the aim to develop and evaluate an ML-based approach for function merging by predicting alignment scores, this project has successfully fulfilled all objectives outlined in section 1.2, demonstrating the following achievements.

1. **Framework and Data Foundation**: Successfully developed a robust framework utilising modified F3M and IR2Vec for data collection, coupled with SQLite for storage, enabling the successful gathering of a large-scale dataset comprising ***2.2 billion*** function pair samples with their alignment scores.

2. **ML Model Development and Evaluation**: Successfully designed and trained two deep learning models (***Dot Product Siamese*** and ***Multi-Headed Self-Attention***) to predict alignment scores from function vector representations, directly addressing the core aim of replacing handwritten-heuristics with ML.

   - The Dot Product model showed reasonable accuracy for scores up to 0.7 but plateaued thereafter.
   - The Multi-Headed Self-Attention model demonstrated superior accuracy across the full range of alignment scores.

3. **Training Efficiency Enhancements**: Successfully implemented data pre-processing techniques, including pre-splitting and serialisation, which significantly accelerated the model training process by up to ***60%***, making the development and evaluation process more feasible.

4. **Improved Code Size Reduction**: Evaluated the ML approach against F3M. While F3M achieved slightly better overall binary size reduction, this is largely due to increased exception handling metadata, over which function merging had less control. Crucially, the ML models achieved a superior reduction in the compiled-code section size (***48%*** improvement in reduction), the reduction over which function merging has the most direct control. This directly demonstrates the effectiveness of the ML approach in optimising the core code instructions, a key aspect of improving function merging.

5. **Enhanced Predictability**: Demonstrated that the ML models offer ***more predictable and safer*** code size reduction behaviour compared to F3M. The Dot Product model never resulted in size increases, and the Attention model only did so rarely, contrasting with F3M's variability, particularly on smaller benchmarks.

6. **Demonstration of ML Potential**: Demonstrated that machine learning can create more effective merging decisions for function merging based on predicted alignment scores without relying on hand-crafted heuristics for the primary merging decision. This validates the project's core hypothesis while identifying practical challenges like compilation time overhead and integration complexities (e.g., handling newly merged functions, metadata impact) for future consideration.

5.3 Reflection

The most significant challenge in this project was managing the enormous scale of the dataset. With 2.2 billion function pairs, the data volume exceeded typical machine learning datasets by orders of magnitude:

- Data operations that would typically be trivial became substantial bottlenecks. Even basic SQL queries to count rows took upwards of five minutes to complete.

- This scale affected two-thirds of the project pipeline, making data collection and model training extremely time-consuming.

This challenge required good planning to utilise my time effectively. Following my supervisor's advice, this project's schedule was carefully structured around the computational time, maximising the system uptime, ensuring the next processing task was prepared before the current one has completed execution. This approach minimised system idle time and maintained continuous progress. This experience emphasised the importance of planning in data-intensive projects. It led to several engineering innovations, including strategically reducing the dataset without sacrificing model quality, serialising data and implementing multithreading for data loading to speed up the training process. These optimisations proved essential for making the project feasible within the time constraints and provided valuable experience in large-scale data engineering.

Beyond personal challenges, several technical insights emerged during implementation and evaluation.

The versioning conflict between IR2Vec and F3M's LLVM toolchain created an unexpected integration barrier. This required a compromise solution where function embeddings were pre-computed rather than generated during compilation. While functional, this limitation meant newly merged functions could not be considered for subsequent merging, limiting the evaluation of this project.

Compile-time performance emerged as a significant trade-off. The ML-based approach requires significantly more processing time than F3M, primarily because it exhaustively evaluates all possible function pairs to find the optimal merging candidates. While this project prioritised optimisation quality over compilation speed, a production implementation would need to address this performance gap by incorporating an early pruning stage.

While our ML approaches achieved superior compiled-code section size reduction (48% improvement over F3M), the overall binary size reduction was slightly less impressive. This disparity stems from a consequence of function merging, as functions combine, they develop more complex flow while needing to maintain the original exception-handling behaviours, which increases metadata requirements.

This observation highlights how the compiled-code section, where function merging has direct influence, showed clear benefits from our ML approach. However, these gains were partially offset by increases in exception-handling metadata. This demonstrates the interconnected nature of compiler optimisations, where improvements in one area can have cascading effects on others.

The enhanced predictability of our ML models represents another significant achievement. Unlike F3M, which occasionally increased code size on smaller benchmarks, our Dot Product model never produced size increases, and the Attention model rarely did so. This consistency offers compelling value for production environments where predictable behaviour may be as important as optimisation potential.

5.4 Future Work

Building on the findings and limitations identified in this project, several promising research directions could further advance machine learning-based function merging optimisation.

Reinforcement Learning The current supervised learning approach could be complemented by a reinforcement learning strategy that directly optimises for merging profitability. This approach would involve developing a more accurate function size calculation model and training an agent to explore the vast space of potential function pairs. The agent would receive negative rewards for unprofitable merges and positive rewards for profitable ones, gradually learning optimal merging policies through interaction with the compiler environment. This data-driven strategy could potentially discover non-obvious merging opportunities that both hand-crafted heuristics and our current supervised models might miss.

Model Enhancement A natural evolution of our current model would incorporate function size awareness directly into the prediction mechanism. An analysis could be performed on the role function sizes play on the profitability of merging. A model that explicitly accounts for both alignment quality and function size could make more informed decisions about which pairs warrant merging attempts, potentially improving both compilation time and optimisation outcomes.

Hybrid Method To address the compilation time overhead identified, a hybrid approach combining F3M's efficient search space reduction with our ML models' prediction accuracy could offer the best of both worlds. This would use F3M's locality-sensitive hashing to identify promising function clusters quickly, then apply the ML approach to functions within the same band to select the optimal merging candidate. This approach would substantially reduce the number of pairs requiring expensive alignment score predictions while maintaining high-quality merging decisions. Alternatively, a new multi-tiered profitable analysis can be developed to reduce the search space for the ML approach.

Error Handling Our evaluation revealed that increased exception-handling metadata partially offset improvements in compiled-code section size. This suggests a valuable research direction explicitly focused on understanding and optimising how function merging affects exception-handling requirements to propose a new merging technique that maintains proper error handling while minimising metadata growth. This targeted approach would help ensure code size reductions translate more directly to overall binary size improvements.

Bibliography

[1] Create and set up your app. https://support.google.com/googleplay/android-developer/answer/9859152?hl=en. Accessed: 2025-4-7.

[2] Dropout — PyTorch 2.6 documentation. https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html. Accessed: 2025-4-8.

[3] The LLVM compiler infrastructure project. https://llvm.org/. Accessed: 2025-4-23.

[4] LLVM language reference manual — LLVM 21.0.0git documentation. https://llvm.org/docs/LangRef.html. Accessed: 2025-4-23.

[5] LLVM: Lib/transforms/IPO/MergeFunctions.Cpp source file. https://llvm.org/doxygen/MergeFunctions_8cpp_source.html. Accessed: 2025-3-17.

[6] LLVM's analysis and transform passes — LLVM 21.0.0git documentation. https://llvm.org/docs/Passes.html. Accessed: 2025-4-23.

[7] MergeFunctions pass, how it works &x2014; LLVM 21.0.0git documentation — llvm.org. https://llvm.org/docs/MergeFunctions.html. [Accessed 18-03-2025].

[8] Tf.Keras.Layers.Dropout. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dropout. Accessed: 2025-4-8.

[9] Hussain Alibrahim and Simone A. Ludwig. Hyperparameter optimization: Comparing genetic algorithm against grid search and bayesian optimization. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 1551–1559, 2021.

[10] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine learning from theory to algorithms: An overview. *Journal of Physics: Conference Series*, 1142:012012, November 2018.

[11] Apple Inc. Maximum build file sizes. https://developer.apple.com/help/app-store-connect/reference/maximum-build-file-sizes/. Accessed: 2025-4-7.

[12] Amir H. Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, and Yaoqing Gao. Work-in-progress: Mlgoperf: An ml guided inliner to optimize performance. In *2022 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, page 3–4. IEEE, October 2022.

[13] Thomas Bayes and David Hume. Bayes's theorem. In *Proceedings ofthe British Academy*, volume 113, pages 91–109, 1763.

[14] Yoshua Bengio, Ian Goodfellow, Aaron Courville, et al. *Deep learning*, volume 1. MIT press Cambridge, MA, USA, 2017.

[15] Christopher M Bishop and Hugh Bishop. Gradient descent. In *Deep Learning: Foundations and Concepts*, pages 209–232. Springer, 2023.

[16] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, page 2787–2795, Red Hook, NY, USA, 2013. Curran Associates Inc.

[17] A.Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997.

[18] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In J. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6. Morgan-Kaufmann, 1993.

[19] Milind Chabbi, Jin Lin, and Raj Barik. An experience with code-size optimization for production ios mobile applications. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 363–377, 2021.

[20] Davide Chicco. *Siamese Neural Networks: An Overview*, pages 73–94. Springer US, New York, NY, 2021.

[21] Ian Colbert, Jake Daly, and Norm Rubin. Generating gpu compiler heuristics using reinforcement learning, 2021.

[22] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, page 253–262, New York, NY, USA, 2004. Association for Computing Machinery.

[23] Tobias J.K. Edler von Koch, Björn Franke, Pranav Bhandarkar, and Anshuman Dasgupta. Exploiting function similarity for code size reduction. *SIGPLAN Not.*, 49(5):85–94, June 2014.

[24] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. Milepost gcc: machine learning based research compiler. *Proceedings of the GCC Developers' Summit 2008*, 06 2008.

[25] Daniel Gibert, Carles Mateu, and Jordi Planes. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.

[26] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning, 2020.

[27] Ibrahem Kandel and Mauro Castelli. The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset. *ICT Express*, 6(4):312–315, 2020.

[28] Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention, 2022.

[29] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 12 1989.

[30] Ioannis E. Livieris, Emmanuel Pintelas, Niki Kiriakidou, and Panagiotis Pintelas. Explainable image similarity: Integrating siamese networks and grad-cam. *Journal of Imaging*, 9(10), 2023.

[31] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[32] E. Mollick. Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75, July 2006.

[33] Felix Mühlbauer, Lukas Schröder, Patryk Skoncej, and Mario Schölzel. Handling manufacturing and aging faults with software-based techniques in tiny embedded systems. In *2017 18th IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2017.

[34] 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.

[35] William F. Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 245–256, 2017.

[36] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[37] Jeanne Pereira and Filipe Saraiva. A comparative analysis of unbalanced data handling techniques for machine learning algorithms to electricity theft detection. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020.

[38] Xin Qian and Diego Klabjan. The impact of the mini-batch size on the dynamics of {sgd}: Variance and beyond, 2021.

[39] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. Hyfm: function merging for free. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 110–121, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 149–163, 2019.

[41] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[42] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[43] Radhey Shyam and Riya Chakraborty. Machine learning and its dominant paradigms. 8:2021, 09 2021.

[44] Enoch Solomon, Abraham Woubie, and Eyael Solomon Emiru. Deep learning based face recognition method using siamese network, 2024.

[45] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[46] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*, pages 123–134, 2005.

[47] Sean Stirling, Rocha Rodrigo C. O., Kim Hazelwood, Hugh Leather, Michael O'Boyle, and Pavlos Petoumenos. F3m: Fast focused function merging. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253, 2022.

[48] Sriraman Tallam, Cary Coutant, Ian L Taylor, David X Li, and Chris Demetriou. Safe icf: Pointer safe and unwinding aware identical code folding in the gold linker. *Proceedings of the 2010 GCC Summit*, pages 107–114, 2010.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[50] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), December 2020.

[51] S. VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. Rl4real: Reinforcement learning for register allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, page 133–144. ACM, February 2023.

[52] Jia Wu, Xiu-Yun Chen, Hao Zhang, Li-Dong Xiong, Hang Lei, and Si-Hao Deng. Hyperparameter optimization for machine learning models based on bayesian optimizationb. *Journal of Electronic Science and Technology*, 17(1):26–40, 2019.

[53] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. Online memory compression for embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9(3), March 2010.

[54] Long Zhang, Yangyuan Liu, Jianmin Zhou, Muxu Luo, Shengxin Pu, and Xiaotong Yang. An imbalanced fault diagnosis method based on tffo and cnn for rotating machinery. *Sensors*, 22:8749, 11 2022.

Appendix A

Appendix

A.1 Acronyms

Acronym	Full-Form
CFG	Control-Flow Graph
ML	Machine Learning
F3M	Fast Focused Function Merging
MSE	Mean Squared Error
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
IR	Intermediate Representation
IOT	Internet of Things
SOTA	State-of-the-Art

A.2 Calculations

Size of Input Sample

$$\frac{32\ bits \times 601\ Elements}{8 \times 1024} = 2.34KB$$

Profitable Percentage

The percentage of merge attempts which are profitable.

$$Profitable\ Percentage(\%) = \frac{No.\ of\ Profitable\ Merges}{Total\ Number\ of\ Merges} \tag{A.1}$$

A.3 Artifacts

Project's Artifacts: `https://github.com/chuongg3/ThirdYearProject`
Modified F3M Codebase: `https://github.com/chuongg3/llvm-function-merging`