Static Branch Prediction through Representation Learning

PIETRO ALOVISI

Master in Computer Science
Date: June 8, 2020
Supervisor: Roberto Castañeda Lozano
Examiner: Magnus Boman
School of Electrical Engineering and Computer Science
Swedish title: Statisk Branch Prediction genom Representation Learning

Abstract

In the context of compilers, branch probability prediction deals with estimating the probability of a branch to be taken in a program. In the absence of profiling information, compilers rely on statically estimated branch probabilities, and state of the art branch probability predictors are based on heuristics. Recent machine learning approaches learn directly from source code using natural language processing algorithms. A representation learning word embedding algorithm is built and evaluated to predict branch probabilities on LLVM's intermediate representation (IR) language. The predictor is trained and tested on SPEC's CPU 2006 benchmark and compared to state-of-the art branch probability heuristics. The predictor obtains a better miss rate and accuracy in branch prediction than all the the evaluated heuristics, but produces and average null performance speedup over LLVM's branch predictor on the benchmark. This investigation shows that it is possible to predict branch probabilities using representation learning, but more effort must be put in obtaining a predictor with practical advantages over the heuristics.

Keywords — compiler, compiler optimization, branch prediction, machine learning, representation learning, LLVM.

Sammanfattning

Med avseende på kompilatorer, handlar branch probability prediction om att uppskatta sannolikheten att en viss förgrening kommer tas i ett program. Med avsaknad av profileringsinformation förlitar sig kompilatorer på statiskt uppskattade *branch probabilities* och de främsta *branch probability predictors* är baserade på heuristiker. Den senaste maskininlärningsalgoritmerna lär sig direkt från källkod genom algoritmer för *natural language processing*. En algoritm baserad på *representation learning word embedding* byggs och utvärderas för *branch probabilities prediction* på LLVM's *intermediate language* (IR). Förutsägaren är tränad och testad på SPEC's CPU 2006 riktmärke och jämförd med de främsta *branch probability* heuristikerna. Förutsägaren erhåller en bättre frekvens av missar och träffsäkerhet i sin *branch prediction* har jämförts med alla utvärderade heuristiker, men producerar i genomsnitt ingen prestandaförbättring jämfört med LLVM's *branch predictor* på riktmärket. Den här undersökningen visar att det är möjligt att förutsäga *branch prediction probabilities* med användande av representation learning, men att det behöver satsas mer på att få tag på en förutsägare som har praktiska övertag gentemot heuristiken.

Nyckelord — kompilator, kompilatoroptimering, branch prediction, maskininlärning, representation learning, LLVM.

Contents

Chapter 1

Introduction

Compilers [1] are programs whose main task is to translate a program written in a high-level language into an assembly one for a particular instruction set architecture. Given the many source and destination languages, compilers split the translation process into two parts: first the source language is converted into an intermediate representation (IR) language, then the IR is transformed into the target assembly language. The former task is performed by the so-called *Frontend* while the latter is accomplished by the *Backend*. The common structure of a compiler is shown in Figure 1.1. Since the IR acts as an interface between frontends and backends, they can be developed independently.

The other goal of compilers is code optimization. Optimizations are introduced as transformations of the code that change its structure but not its semantic. The optimizations can target, for example, execution speed, power consumption, or code size. Optimizations can be of two types: machine-dependent and machine-independent optimizations. The former are handled by the backend and exploit specific features of the target architecture, while the latter are performed at the IR level by the so-called *middle-end* and are target independent.

Crucial to most optimization are the parts of the code that gets executed the most, called *hotspots*. Since they represent a great share of the execution time, their optimization has a greater impact than optimizing sequences of instructions executed less frequently. These regions are often found in loops, or in procedures that are recurrently performed. It is therefore important to identify hotspots before the optimization takes place. To identify the hotspots one has to predict the *control flow* of a program, that is the sequence in which instruction are executed while running. The task is not trivial because the order of the instruction in a program is not the same as the one while running.

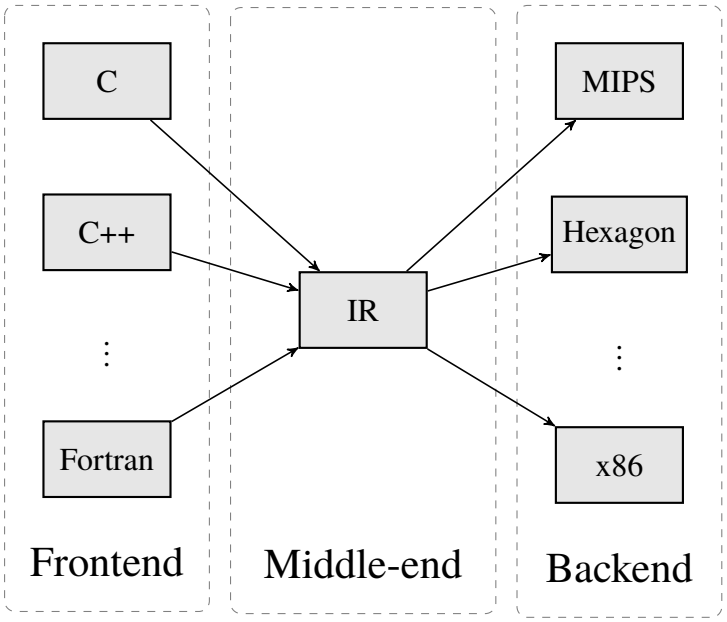


Figure 1.1: Common structure of a Compiler. Each of the three sections is in charge of a translation between two languages.

All imperative languages have built-in constructs to change the control flow such as : `if-else` statements, `for` loops, and `gotos`. These constructs, when transformed into lower-level languages, are represented by the so called *branch instructions* (or just *branches*) that select at runtime the next instruction to perform. Usually branches have only two possible successor instructions (or directions), called *target* and *fall-through* or *taken* and *not-taken*. Predicting the most taken direction of a branch without executing the code is called *static branch prediction*, while assigning a probability to each direction is called *static branch probability prediction*. It has been shown that the execution frequency of regions of code (hence hotspots identification) and branch probabilities are closely related: given one the other can be derived [90, 69]. Therefore a precise prediction of branch probabilities entails an accurate starting point for optimization.

Predicting branches was originally concerned with reducing misprediction in pipeline processors [54]. First the work by Wall [91] and then the one of Fisher and Freudenberger [29], proved that programs' branches have regular behaviour and it is worthwhile to predict them statically (that is without executing the code). In the 1990s heuristics [4, 96, 90] were developed to perform static branch probability prediction. The probability values were chosen

by averaging many profiled programs' behaviours. Their results were already promising obtaining a dynamic miss rate of around 30% against a theoretical limit of around 10% on the tested programs. At the turn of the century few machine learning methods were also employed for branch prediction [14, 12]. In these approaches hand-crafted features were extracted both from the program structure and the instructions, next a machine learning algorithm classifies automatically the branch as taken or not taken. These methods result in better performance than the heuristics developed before. The downside is that these methods do not predict branch probabilities but only the most taken direction. These new techniques are of no use for current compilers such as LLVM [45] or GCC [85] which use branch probabilities in their internal representation to perform optimization. For this reason, their static estimation still relies on the older heuristics.

In the last two decades research attention has shifted from static branch prediction to *dynamic branch prediction* [59, 43, 75], whose goal is predicting at runtime the direction taken by a branch. Dynamic branch prediction is very important to reduce performance penalties with speculative execution in CPUs [37]. Recently applications of machine learning have spawned in tasks concerning source code inputs [93, 3], examples are: power consumption estimation of programs [56], optimization selection in compilers [23, 30], bug identification and text to source code translation [3]. In the literature, a common way of dealing with source code inputs is using natural language processing techniques [3] both for representing it and for prediciton methods.

Problem statement Despite the various accomplishments of Machine Learning in various compiler-related areas, there is no record of its application in branch probability predictor and its effect on compiler optimization.

Solution outline Branch prediction need to be done at the IR level so that the various optimizations can harness branch information. Taking inspiration from recent work in [56, 23, 30] the branches will be predicted using neural networks [10, 60] with features extracted automatically from the IR using *Representation Learning* [32]. Representation Learning builds a representation of the input without any human-designed feature extraction. The goal is to let the algorithm choose which combination of features is best for the task and avoid loss of information due to human design. This is motivated by its successful application in other compiler related task [52, 56, 23].

1.1 Research Question

Branch probabilities are often the starting point for program optimization in compilers. The research community has not yet explored the potential of current machine learning approaches leaving possible improvements unexploited. The main question this project tries to answer is: *what is the benefit of representation learning based static branch probability predictor compared to previous approaches?*

Hypothesis The hypothesis is that a representation learning algorithm performs better than current heuristic, both as a branch predictor and as a branch probability predictor.

Purpose An accurate prediction of branching probabilities is useful for performing various compiler optimizations [1, 84] and probabilistic dataflow analysis [68]. Current production compiler use simple heuristic which can be improved. The purpose of the research question is to assess the benefit of representation learning and machine learning for branch probability prediction compared to state-of-the-art heuristics. Another target for this project are aggressive compilers such as UNISON [50] that embeds branch probabilities in its objective function for register allocation and instruction scheduling. Wrong prediction of branch probabilities might guide the optimization into a suboptimal solution.

Goal The goal is to devise a machine learning algorithm to perform static branch probability prediction given the IR of a program.

Benefits, Ethics and Sustainability Since static branch prediction is the basis for many optimization, the possible benefits are numerous. A compiler might use the information to optimize a program for speed or for energy consumption. The cost for performing branch prediction once during compiling is amortized during the execution of the program, even more if the program is distributed across many devices.
Both the execution and the result of the project do not pose any ethical problem. In a society ever so reliant and pervade by computer systems, their program's optimization is crucial both for speed and for energy consumption [5]. These arguments motivate the research for a social, economical and environmental sustainability.

Method and Methodology The research question will be addressed using a quantitative method. The problem will be analyzed on the intermediate representation language of the LLVM compiler infrastructure [45]. A data-driven branch predictor will be constructed from branching information collected from SPEC CPU 2006 Benchmark [38], a benchmark widely used in previous branch prediction research [90, 4, 12]. The evaluation of the branch predictor and its comparison with the state-of-the-art heuristics follows an experimental method: the accuracy of the predictors and their effect on performance are collected and analyzed.

1.2 Contributions

This thesis gives the following contributions:

- An analysis of the branch predictor implemented in LLVM and its implication on performance;
- An implementation of Ball and Larus [4] and Wu and Larus [96] heuristics into LLVM's framework;
- A comparison between the above heuristics with LLVM's branch predictor;
- A machine learning model to predict branch probabilities at LLVM's IR level of a compiler.

1.3 Document Outline

Chapter 2 gives an introduction to the concepts used in the dissertation along with the relevant literature study for the subject. Chapter 3 performs an analysis of the current branch predictor in LLVM and compares it with state-of-the-art heuristics. Chapter 4 continues with the description of the branch probability predictor developed in this thesis, it also highlights the design decisions, the dataset used, and the evaluation methods. The results are presented in Chapter 5 and discussed in Chapter 6 outlining the conclusion of the project and the future work.

Chapter 2

Background

This chapter introduces the concepts necessary to understand the content of the thesis. It begins presenting the Control Flow Graph in Section 2.1, its relation with branch probabilities, and hotspot identification in programs. Section 2.2 introduces Machine Learning, its core concepts and applications to compiler technologies. The Chapter continues with a display of the relevant literature in Section 2.3. Finally Sections 2.5, 2.4, and 2.6 present necessary machine learning models used later in the dissertation.

2.1 Control Flow Graph

Imperative programming languages are equipped with *conditional constructs* such as `for` loops and `if-else` statements. At runtime, these statements select which instruction sequence to run based on the outcome of binary condition. In a program's execution, the sequence of instructions performed is called the *control flow* of a program and depends on the input of the program. Conditional constructs are represented at a lower-level language through *branch instructions* (often just *branches*). Similarly to conditional constructs, branches determine the next instruction during an execution. There exists two types of branches: conditional and unconditional. Conditional branches establish the next instruction according to a boolean condition, while unconditional do not. An example of a conditional branch is given by the instruction `breq` ('branch on equal') in the following snippet of pseudo-assembly.

```
...
sub  a, b, 3          # a = b - 3
breq c, 0 , %target   # c == 0
```

```
add  a, a, c            # a = a + c
...
%target:
add  a, a, 1            # a = a + 1
...
```

The instruction checks if `c` is equal to `0`. If the test is false the execution continues normally with the succeeding instruction (called *fall-through* or *branch not-taken*), otherwise the execution is resumed at another program point described by the `%label`(called *target* or *branch taken*).

On one hand, branches are what makes programming languages Turing complete [11], on the other hand, they complicate some compiler tasks such as register allocation, instruction scheduling and, optimization, as the compiled program must run properly no matter which order of instruction is executed [1, 84]. To solve these problem compilers represent procedures and programs as Control Flow Graphs (CFGs), a structure that expresses all possible program's control flows. The formal definition of the CFG starts from the concept of basic block.

Definition 2.1 *A basic block is a maximal set of adjacent instructions that are always executed in sequence.*

The successors of a basic block bb_i are the basic blocks that can be executed immediately after bb_i, and are represented by $succ(bb_i)$. Successors form a binary relation over the set $\mathcal{B}$ of all basic blocks in a program: let $\rightarrow \subseteq \mathcal{B} \times \mathcal{B}$ denote in infix notation the following relation $bb_i \rightarrow bb_j \iff bb_j \in succ(bb_i)$.

Definition 2.2 *The Control Flow Graph (CFG) is a directed graph $\mathcal{G} = (\mathcal{B}, \mathcal{E})$ whose vertices $\mathcal{B}$ is the set of basic blocks $bb_1, bb_2, \ldots, bb_N$ and an edge $e_{i,j} = (bb_i, bb_j) \in \mathcal{E}$ iff $bb_i \rightarrow bb_j$. Moreover there exist a node $bb_e \in \mathcal{B}$ which is called the entry node.*

In other words, the CFG is the graph that represents the successor relation over the basic blocks of a program. The CFG has an initial node and (possibly multiples) exiting nodes: the initial node is the entry point of the procedure while exiting nodes are those which contain the exit point of the procedure. An example of a CFG is shown in Figure 2.1. Here the entry node is highlighted in green, while the exiting node is red. In yellow is highlighted a basic block terminating with a conditional instruction, and therefore is has two successor

basic blocks. The CFG is an essential data structure in compilers as it is used to perform static analysis, code transformation, and code optimization [70, 1, 84].

```
void function(){
    c = a + b;
    a ++;
    while(c < N){
        c = c - b;
        a ++:
    }
    print(c);
    return;
}
```

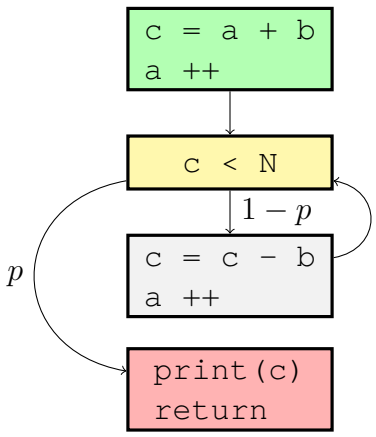


Figure 2.1: Correspondence between source code (on the left) and its Control Flow Graph (on the right). The green basic block is the entry node of the CFG, while the red one is the exit node. In yellow a basic block containing a conditional branch instruction is highlighted.

Many optimization passes such as code inlining and block reordering need information about the most executed blocks and most taken direction of branches. These two information are tightly coupled as later explained in Subsection 2.1.1. They can be obtained into two ways: via profiling the program, which requires running it, or estimate it statically without running it.
The former is usually believed to be expensive to carry out, and therefore estimation methods are more often used. If the latter is an accurate approximation of the former, it is possible to avoid profiling entirely.

Static Branch Prediction (SBP) refers to the prediction of the most taken direction of a branch, while *static branch probability prediction* (SBPP) estimates the probability of each possible successor of a basic block. They are called "static" because the predictions are performed without running the program. From a SBPP is possible to obtain a SBP by predicting the successor with the highest probability to be the most taken direction. Referring to Figure 2.1 and the yellow basic block, a SBPP estimates the quantities p and $1 - p$, while a SBP is concerned in predicting whether $p > 1 - p$ or not. The *miss rate* is a widely used metric for evaluating branch predictors. It is defined as the percentage of the execution of branches that are mispredicted by the SBP. To obtain a lower bound on the miss rate for a given program and input, it is

possible to use the profile of the program as a branch predictor on the same input. This is called the Perfect Branch Predictor (PBP) [4]. Subsections 2.3.1 and 2.3.2 presents the literature review for SBP and SBPP.

2.1.1 CFG as Markov Chains

Markov Chains [60, 34] are a stochastic model used for describing sequences statistically. They are often used to model walks in a directed graph. In the context of compilers, they can be used to model the CFG and branch probabilities altogether. With this representation, it is possible to compute the average execution frequency of each basic block.

Discrete-time Markov Chains (DTMC) are defined as a series of random variables $X = \{X_i\}_{i=1,\dots,T}$ that obey the Markov property. The Markov property postulates that the distribution of every random variable depends only on the outcome of the previous one, as described in the following equation:

$$p(X_i|X_{1:i-1}) = p(X_i|X_{i-1}) \tag{2.1}$$

Where the notation $X_{i:j}$, with $i \leq j$, means all the random variables from index i to index j included.

Markov Chains that assume that the conditional probability $p(X_i|X_{i-1})$ does not vary on the index i are called *homogeneous*. If each variable can take a finite number of values $X_i \in S = \{S_1, \dots, S_N\}$ and homogeneity is assumed, it is possible to represent a Markov Chain through its state space. This involves representing a graph whose nodes are the elements of S and whose edges occur between two nodes $i \to j$ if and only if $p(X_t = j|X_{t-1} = i) > 0$. Each edge is also labelled with their corresponding probability $p_{i,j} = p(X_t = j|X_{t-1} = i)$, referred as the transition probability between the two states. An example is shown in Figure 2.2. The graph can be furthermore described by a square matrix, the *transition matrix*, whose entries are $T\{i, j\} = p_{i,j}$. A realization of the sequence X of random variables represents a walk in this graph.

Relation to CFG Control Flow Graphs are easily mapped into Markov Chains: each state represents a basic block ($S = \mathcal{B}$), and an execution of a program is a sequence of basic blocks. The transition probability $p_{i,j}$ between two states is no more than the branch probability between two basic blocks. If we assume that the Markov property of Equation 2.1 holds then it is possible to calculate the average number of times each state is visited, or equivalently the percentage of "time" spent in each basic block. Given the transition matrix, the vector

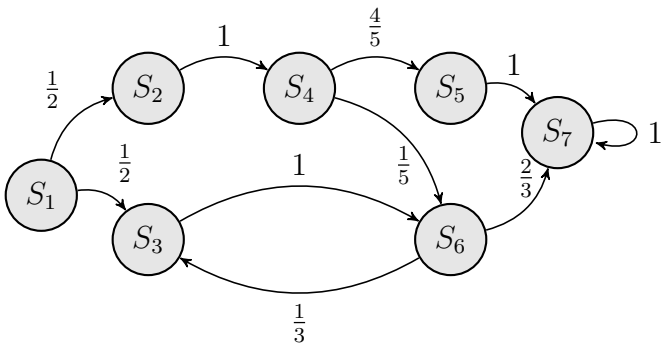


Figure 2.2: Example of a state space of a Markov Chain. An example of a walk in the graph is the sequence $X = [S_1, S_2, S_4, S_6, S_3, S_6, S_7]$.

of the frequency in each state $\mathbf{w}$ is given by solving the following equation:

$$(\mathbf{T} - \mathbf{I})^T \cdot \mathbf{w} = \mathbf{0} \tag{2.2}$$

Hence the vector of frequencies is the eigenvector of the matrix $(\mathbf{T} - \mathbf{I})^T$. Equation 2.2 is well defined when the chain is *ergodic*: if from any state it is possible to reach any other state. Control flow graphs are not ergodic because terminating basic blocks do not have any successors, but can be made such by: removing unreachable basic block, and by artificially adding edges from each terminating basic block to the entry one.

Given branch probabilities, it is possible to identify which basic blocks are the most executed in a procedure, hence hotspot identification. Wagner *et al.* [90] computes the frequency of each state using an iterative approximation of equation 2.2 while also suggesting how this approach can also be applied to the function call graph [84]. The result is a estimate of how many times a function gets called, another useful information for compiler optimization.

2.2 Machine Learning

The following paragraphs introduce the area of machine learning and its key concept. After the related work presented in Section 2.3, Sections 2.4 and 2.6 introduce the models used in the rest of the thesis.

Machine Learning [10, 60] is a set of models and algorithms to automatically identify and exploit patterns in data. The goal of every machine learning technique is at its core an optimization problem and can be summed up as follows: from a dataset $\mathcal{D}$, called *training set*, learn the parameters θ of a family

of functions $f(\theta, \cdot)$ such that the value of another function, called the *loss* $\mathcal{L}$, is minimized:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(f(\theta, \cdot), \mathcal{D}) \tag{2.3}$$

The loss function measures how well the function $f(\theta, \cdot)$ fits the data. The loss can arise statistically or can be created at will. The parameters θ^* can be found exactly or can be iteratively searched. The function $f(\theta^*, \cdot)$ is the learned model and it is evaluated on another dataset called the **test set**. The test set is used to check if the model has learned to perform the correct task on another dataset other than the training one.

Applications Even though Machine Learning dates back to the early 1940's [53], only in the last decade new sophisticated models were devised due to two factors: the computational capabilities of computers and the high quantity of available data [60]. Among the latest developments in machine learning is Deep Learning [32], a set of techniques that brought significant improvements in fields such as computer vision [46] and natural language processing [67]. Machine learning is used in many contexts, for example in physics [36], in healthcare [28], music composition [62] and many more.

For code and compilers Recently, machine learning techniques have found their use in compilers, where they are used mainly for two tasks: estimating statically dynamic features or performing optimization of the code. Examples of the first are *Ithemal* [56] which is able to predict a basic block's throughput in cycles, or CPU's power consumption [9]. For the task of optimization, machine learning methods have been used for choosing which optimization to perform [18], or for choosing the level of parallelism of a piece of code [51]. The review by Wang and O'Boyle [93] presents the areas and the techniques used in the literature for combining machine learning for compiler optimization.

Other uses of machine learning techniques on program sources are related to software engineering tasks, such as: classifying programs [76], identifying test cases [17], or finding program inputs that can undermine the program's security. For these tasks techniques similar or inspired by natural language processing are used to extract features [3]. Other types of representation take advantage of Graph Neural Networks [97] as described by Allamains *et al.* [2] or as used in the work of Shi *et al.* [81].

2.3 Related Work

Subsection 2.3.1 presents the heuristic approaches in branch prediction, while subsection 2.3.2 shows the use of machine learning for the same problem. Finally subsection 2.3.3 presents the techniques currently used in compilers for branch prediction.

2.3.1 Heuristics

The early work focused mostly on estimating program execution time by using branch probability. In the work of Ramamoorthy [69] each function is associated with a mean execution time, and the flow of the program is represented as a directed graph whose arcs are labeled with the probability of switching from a function to another. From this graph, it is possible to compute mean and variance of a program's execution time as shown by the paper of Ramamoorthy itself. His main contribution is in modeling the program flow as a Markov Model. In the paper, there is no mention of ways in which the probability of each branch is computed other than profiling. The same objective is shared by the work of Sarknar [73]. In his work he presents a statistical framework for determining the mean and variance of a program execution by using profiling information. His framework is based on the interval structure [1] of the CFG instead of relying on a Markov Model.

The most prolific years for static branch prediction were perhaps the 1990s. The possibility of predicting statically branches was inquired by Fisher and Freudenberger [29], whose investigation dealt with finding how predictable are branches given a previous execution of a program. They showed that across executions, branches take the same direction most of the time. Wall [91] addressed two crucial questions for program profiling: how reliable is the information taken from one profile run to other runs, and how good can a static estimator be. He looked at various profiled metrics, among which basic blocks frequency. He compared some simple heuristics with the profile information. He concluded that is worthwhile to use profile information for performing optimization and other types of analysis, and that the static predictors he proposed were substantially inferior.

The first heuristics were proposed by Ball and Larus [4]. They proposed two simple heuristics for predicting the most common branch from static features: one works on branches involved in loops, and the other work on non-loops branches [1]. The heuristic of non-loop branches is a combination of 7 other rules that analyze some static features of the branch. The 7 heuristics are

tested one after the other, and only the first one that applies is considered. Each branch is assigned with a probability estimated from common probabilities in profiling data. The algorithms were tested on MIPS assembly code.
Wagner *et al.* [90] developed another heuristic that assigns a probability to each branch. Their approach worked both at the inter- and intra-procedural level. Their analysis uses both information from a simplified Abstract Syntax Tree and Control Flow Graph for each function as input. They treat differently loops prediction and branch prediction. They used the Markov model to identify the most executed parts of code and estimate function frequency. In their analysis, they conclude that their branch prediction does not improve much the loop prediction of Ball and Larus [4]. Like Ball and Larus's heuristic, breaches are assigned with hard-coded branch probabilities.
Wu and Larus [96] in the same year proposed an improvement on Ball's and Larus's work [4] by incorporating Dempster-Shafer theory [98] to combine the results of the different heuristics. Their results are significantly better than the original one.
Wong [95] developed heuristics for predicting statically branches at the source level. He propose some heuristics and compare them with the perfect static branch predictor. They obtained a miss rate of around 20% compared with the 10% for the PBP. Their conclusion is that semantic information is very valuable for static branch predictor. Finally, Sheikh *et al.* [77] propose a mixed hardware and software approach to reduce branch misprediction to avoid.

2.3.2 Machine Learning Approaches

In 1995 Calder *et al.* [13, 14] applied neural networks to perform branch prediction. They extract statically a set of features from the assembly code to feed into the network. The downside of their approach is that it does not assign branch probabilities, but it classifies the brancha s taken or not taken. The approach of Calder *et al.* has influenced other works: Liu *et al.* [48] perform branch prediction for estimating the cost of reordering basic block for certain substructures of the CFG. In 2002 Sherwood *et al.* [80] tried to automatically classify program's behavior by clustering the distribution of time spent in each basic block, called Basic Block Vectors [79]. They show how it is possible to identify during the execution of a program similar program behavior intervals. They argue that these intervals are useful for program simulation in computer architectures, especially in finding simulation points that represent the behavior of the program as a whole. They show how to find these points and evaluate their accuracy, later implemented in the *SimPoint* program [64].

Veerle *et al*. [26] extended the work of Ball and Larus [4] proposing a mixed approach by including the work of Calder [14] to obtain a better branch predictor. They concluded that this hybrid approach is beneficial though limited by the assembly language used in the dissertation.

Recently Buse *et al*. [12] developed a very accurate model for predicting path frequency using logistic regression in Java programs. They set up the problem as classifying paths in a method as high or low frequent. Their work exploits source level features and the structure of object-oriented programming languages. They also used path frequency for creating a branch predictor, and compare it with Ball's and Larus's, obtaining better results in almost all test benches considered. The downside is that it uses object-oriented source level features, that are not easily applicable to other languages. A comparison between the result of the different approaches but Buse's one appears in [14] and it is reported in Table 2.1.

Method	**Miss Rate (%)**
Backward Taken, Forward Not Taken	34
Ball's and Larus' [4]	26
Wu's and Larus' with Dempster-Shafer theory [96]	25
Calder's *et al*. [14]	20
Perfect branch predictor	8

Table 2.1: Comparison of the different branch prediction schemes in terms of branch miss rate on different benchmarks. The lower the score the better. The results come from [14].

Another related work is that of Tetzlaff and Glesner [89]. Their goal was not to predict branches but the number of iterations of a loop. They employed decision trees [60] fed with more than 100 static features of the loop.
The use of machine learning has also influenced dynamic branch prediction [75, 43] where most of the recent research has focused on. Shi *et al*. [81] proposed a mixed approach for dynamic branch predictor which uses static information in the form of a Graph Neural Network [74] to aid the dynamic prediction.

2.3.3 Branch Prediction in Compilers

This subsection presents the schemes used in two prominent compilers used nowadays: GCC [86] and LLVM [45]. It is notable that both have a way for the programmer to label the expected branch by the use of the function

`__builtin_expect(long exp, long c)`. Its semantics is that `c` is the expected value of the expression `exp`.

LLVM The following information is only relative to LLVM version 10.0.0. There are two main branch prediction schemes in LLVM. The default one is just assigning a uniform probability to each successor of a basic block. This is done automatically and no pass is called.
The implemented branch prediction works at the LLVM IR level and it is a subset of the one described by Ball and Larus [4] plus some language dependent heuristics. One heuristic after the other are tried in sequence, until one's deemed applicable and the predicted probability is assigned to the branch. The branch predictor is implemented as a pass that visits the CFG in post-order applying the sequence of heuristics shown in Table 2.2.

The detailed implementation of each heuristic can be found in the class `BranchProbabilityInfo` at the method `calculate`[1]. The pass is called multiple times for levels optimization `-O1`,`-O2` and `-O3` as an input for subsequent optimization. More information are detailed in Subsection 3.3. LLVM estimates the frequency of the Basic Block in a function using the predicted branch probabilities. This is done by the class `BlockFrequencyInfo`[2]. The algorithm is an approximation of the Markov chain results in Eq. 2.2 that runs in linear time complexity(in the sum of edges and nodes of the CFG).

GCC GCC has similar heuristics to LLVM, but it also uses many heuristics that are also tailored for the input language, in fact there are heuristics for C and Fortran [3]. There are two different schemes of aggregation of the different heuristics: only the first heuristic that matches is kept, or using Dempster-Shafer theory [98]. In GCC's source [4] are also cited three of the already mentioned papers [4, 96, 13] but there is no mention if and how these papers are used. In particular the one by Calder [13] which uses a neural network, seems not to be implemented, while the other two are implemented in the form of the heuristics and aggregation using Dempster-Shafer.

[1]The class can be found at the following link `https://llvm.org/doxygen/classllvm_1_1BranchProbabilityInfo.html`

[2]The class can be found at the following link `https://llvm.org/doxygen/classllvm_1_1BlockFrequencyInfo.html`

[3]The complete list can be found here `https://github.com/gcc-mirror/gcc/blob/master/gcc/predict.def`

[4]`https://github.com/gcc-mirror/gcc/blob/master/gcc/predict.c`

Heuristic	Description
`MetadataWeights`	Use metadata from profiling or other sources.
`InvokeHeuristics`	If the branch instruction is an invoke, branch taken in likely.
`UnreachableHeuristics`	Branches that lead to unreachable basic blocks are unlikely to be taken.
`ColdCallHeuristics`	A block post-dominated by a block with a call to a cold function is unlikely to be reached.
`LoopBranchHeuristics`	Assign to backward edges high probability, and to exiting edges low probability.
`PointerHeuristics`	Pointer Heuristic from [4]. Applicable if there is a pointer comparison.
`ZeroHeuristics`	Similar as Pointer Heuristic, but with integer comparison.
`FloatingPointHeuristics`	Predict branch based on comparison operator between two floating-points variables.

Table 2.2: Heuristics used by the LLVM compiler framework to estimate branch probabilities. They are evaluated in the order they are shown here.

2.4 Neural Networks

Originally inspired by the biological behaviour of the neurons, neural networks [10] are a class of machine learning models. Their fundamental constituent is the artificial neuron. The artificial neuron computes a non-linear function σ of a linear function of its inputs $x_1, x_2, \ldots, x_n$. The coefficient of the linear functions are the free parameters of the model, which are tweaked during learning. The neuron is depicted in Figure 2.3 and represents the following function:

$$out = \sigma\left(\sum_{i=0}^{n} w_i \cdot x_i + b\right) \tag{2.4}$$

Where the *weights* w_i and the *bias* b are the free parameters. As their name

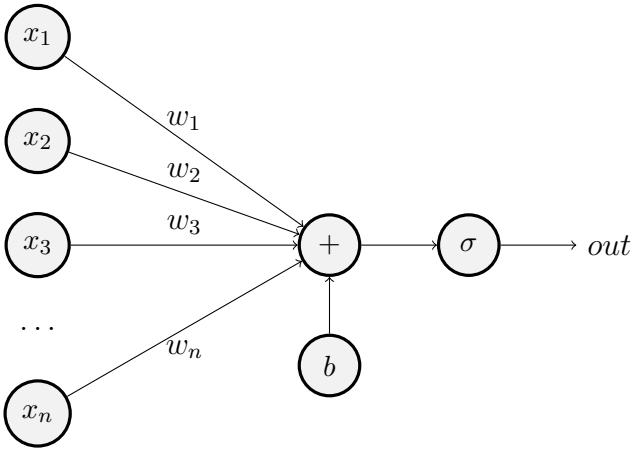


Figure 2.3: The artificial neuron: the scalar variables $x_1, x_2, \ldots, x_n$ are the inputs that get multiplied by the weights $w_1, w_2, \ldots, w_n$. After summing them up, they are further applied to a non-liner function σ.

suggests, neural networks are a set of interconnected artificial neurons. Due to the many possible network shapes and neuron variations, neural networks come in numerous flavours. The following subsections present only two types of neural networks relevant for the rest of the thesis.

2.4.1 Feed Forward Neural Networks

Feed Forward Neural Networks (FFNN) are organized in layers of neurons: the output of a neuron is an input of each neuron in the next layer. The last layer is called the output layer, and its neurons' results is network outcome. The first layer, referred as the input layer, is not made of neurons, instead its components are the entries of the input vector $\mathbf{x}$. All the other layers are called hidden layers, and are composed of a variable number of artificial neurons. The total number of hidden layers is also a free parameter of the network.

An example of a FFNN with only one hidden layer is given in Figure 2.4. The layer structure of neurons allows to conveniently compact the numerous equations of the form 2.4 into Equation 2.5 with the use of vector operations. Bold lowercase symbols represent vectors, while bold uppercase symbols represent matrices. The symbol $\cdot$ represents the matrix multiplication operator.

$$\begin{aligned} \mathbf{z} &= \sigma(\mathbf{W_1} \cdot \mathbf{x} + \mathbf{b_1}) & \mathbf{x} \in \mathbb{R}^N\ \mathbf{z}, \mathbf{b_1} \in \mathbb{R}^H\ \mathbf{W_1} \in \mathbb{R}^{H \times N} \\ \mathbf{y} &= \sigma(\mathbf{W_2} \cdot \mathbf{z} + \mathbf{b_2}) = output & \mathbf{y}, \mathbf{b_2} \in \mathbb{R}^O\ \mathbf{W_2} \in \mathbb{R}^{O \times H} \end{aligned} \tag{2.5}$$

The Universal approximation theorem [41] proves that a Feed Forward Neural Network can approximate, under some weak assumptions, any continuous function with arbitrarily precision, making neural networks a flexible and powerful tool in many scenarios.

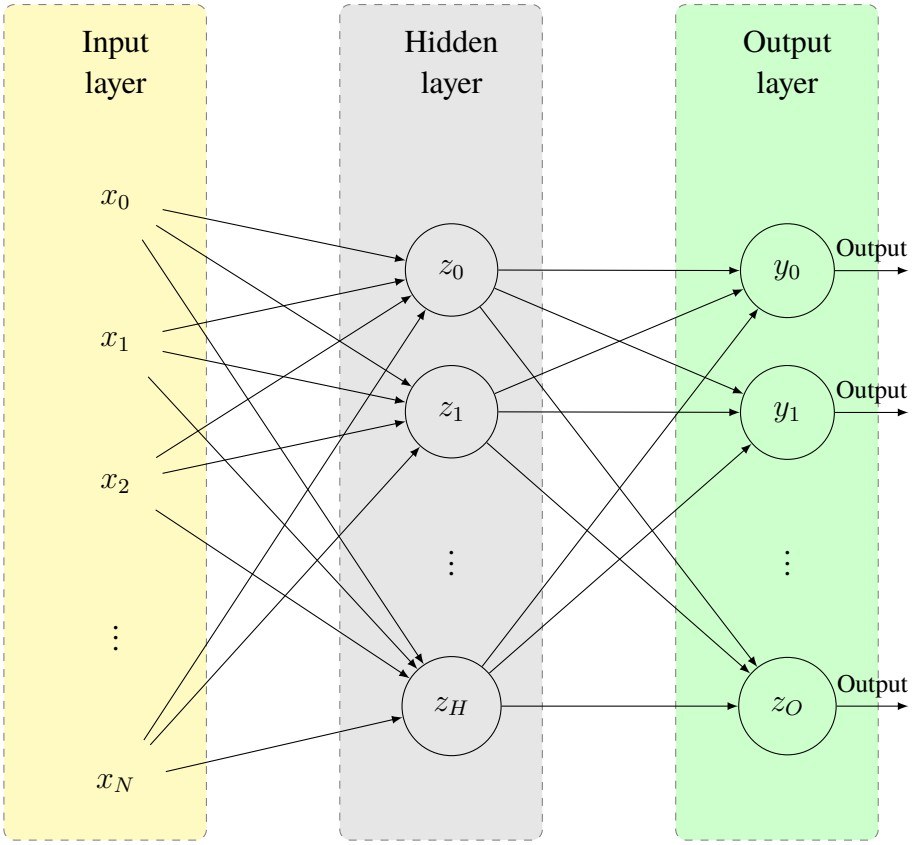


Figure 2.4: Schema of a Feed Forward Neural Network with a single hidden layer. The input layer has size N, the size of the input vector. The output layer is constrained to have size O, the same of the output vector. The hidden layer's size H can be chosen at will.

2.4.2 Recurrent Neural Networks

Recurrent Neural Network (RNN) [32, 78] are a class of neural networks that are designed to deal with sequential input data, such as time series or words

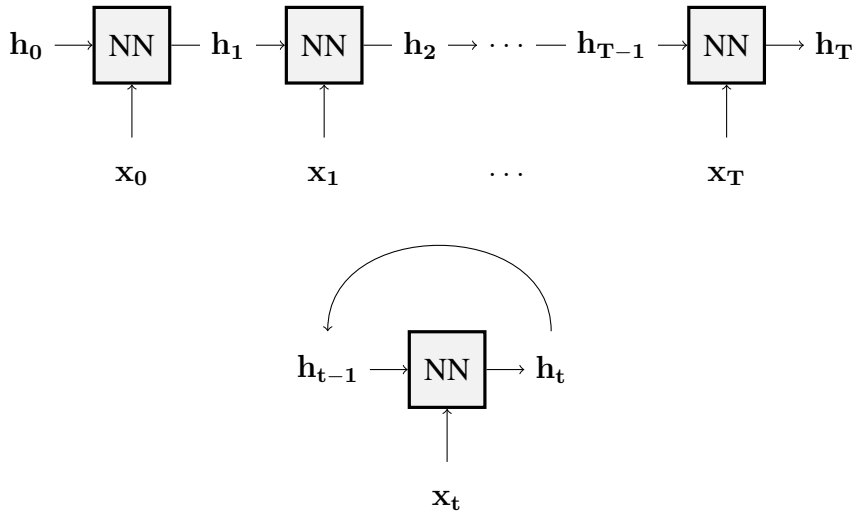


Figure 2.5: How RNN process the input data $\mathcal{X} = [\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_T}]$. The same functions is applied at every input vector $\mathbf{x_i}$ and state vector $\mathbf{x_i}$. The output is the last state vector. The top picture is the "unrolled" version of the bottom one.

in a text. The following paragraph describes a simple RNN architecture. Numerous variations of RNNs develops from this blueprint.

Basic RNN Given and input sequence $\mathcal{X} = \{\mathbf{x_t}\}_{t=0,\ldots,T}$ the idea of RNNs is to compute a vector $\mathbf{h_t}$ (usually called *state*) which encodes the previous inputs. RNNs learn a function that given an input and the previous state outputs the next state. This function is applied sequentially to each input vector. This process is depicted in Figure 2.5.
Given an input sequence $\mathcal{X} = \{\mathbf{x_t}\}_{t=0,\ldots,T-1}$ of vectors of dimension d, the simplest RNN is described by the following equation:

$$\mathbf{h_{t+1}} = \sigma(\mathbf{W} \cdot \mathbf{x_t} + \mathbf{U} \cdot \mathbf{h_t} + \mathbf{b}) \tag{2.6}$$

Where h_t is the state at timestep t, $\sigma(\cdot)$ is a non-linear activation function, such as the hyperbolic tangent or the sigmoid function. $\mathbf{W}$ and $\mathbf{U}$ are matrices, and $\mathbf{b}$ a vector. The output of the RNN is the T-th state $\mathbf{h_T}$. learning finds a value for $\mathbf{W}$,$\mathbf{U}$, and $\mathbf{b}$.

RNNs were developed to overcome the limitations of Time Delay Neural Network (TDNN) and Elman Network [60] which cannot learn long term dependencies due to vanishing gradient [32].

2.5 Code Naturalness

In the area of machine learning and programming language it is often assumed the *naturalness hypothesis* of programming languages [3] which states:

> *"Software is a form of human communication; software corpora have similar statistical properties to natural language corpora and these properties can be exploited to build better software engineering tools."*

Under this assumptions it is possible to use methods of Natural Language Processing with any programming language and therefore LLVM's IR. If this claim holds, the techniques for natural language processing [100, 16] work on source code. Among the applications are code defects identification [19, 101], code to text transformations [42, 61] and vice versa [102, 99], and also program optimization [23].
The next Section presents a type of NLP word embedding technique that is used to encode the LLVM IR statements later in the thesis.

2.6 Word Embedding and `word2vec`

Natural Language Processing (NLP) deals with textual inputs, mostly language used by humans. Originally [20, 15] techniques that work directly on the text were used, with the advancement of ML techniques the prospect changed: it was evident that the two fields could be merged, but there was the need to encode the input to be used by the ML algorithm like the neural network described in Sections 2.4.1 and 2.4.2. The connection can be done with word embeddings: given a vocabulary V of all the possible words, word embedding represents words with vectors.

A simple encoding method is the one-hot encoding which becomes unpractical as the size of the vocabulary increases. To lower the dimension of the encoding Distributed Representation were invented. Distributed Representation represents an input using many dimensions of the embedding space. A common way to deal with these data is to embed each word into a vector, which is a more practical way representation for many tasks.

Among the different types of word embedding techniques *distributed representations* [46, 63, 57, 7] are the most commonly used due to their better generalization and efficiency on large vocabulary of words. Some examples of these embedding algorithms are: [63, 8].

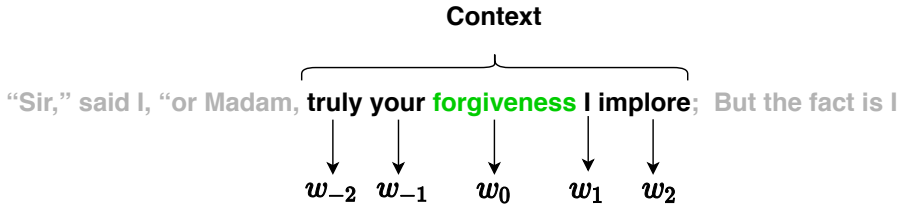


Figure 2.6: The words in black font are the context of the green word "forgiveness" in this excerpt from Edgar Allan Poe's *Raven*[66].

2.6.1 `word2vec`

Mikolov *et al.* [58] proposed two new models for embedding words in a dense vector space called Continuous Bag of Words (CBOW) and Skip-gram model. They are both implemented in the `word2vec` application which is used to indicate them. Both models reduce training time compared to previously proposed algorithms [8].

Both models are fed with a text T and encode words based on the *context* in which they appear in. Mikolov *et al.* define the context of a word $Ctx(w)$ as the set of its surrounding words as exemplified in Figure 2.6, but other definitions can apply. Next the Skip-gram model is described as it is used later in the thesis.

Training Given a set of possible words called the vocabulary V, each word $w \in V$ has two vectorial representations in $\mathbb{R}^N$: v_w called the input vector, and v'_w called the output vector. These vectors are learned to maximize the likelihood of the dataset, as described by the following equation:

$$\arg\max \left\{ \prod_{w \in T} \prod_{c \in Ctx(w)} p(c|w) \right\} \tag{2.7}$$

Where the conditional probability of a word c of being in the context of w is given by:

$$p(c|w) = \frac{e^{v'_c \cdot v_w}}{\sum_{d \in V} e^{v'_d \cdot v_w}} \tag{2.8}$$

Finally all output vectors are discarded, and the input vectors are the representations of each word.

All the process can be conveniently cast into matrix multiplications, and further optimizations can speedup the calculation of the denominator of the

probability in Equation 2.8. `word2vec` is further analyzed in [71, 31].
The `word2vec` model is used in other fields, among them speech recognition (`speech2vec` [21]), graph embedding (`graph2vec` [35]) and biology (`gene2vec` [27]). It is used later in the thesis to encode the statements in the LLVM IR. The explanation is left for later in Chapter 4.

Chapter 3

Analysis of LLVM's Branch Predictor

This chapter analyses LLVM's branch prediction on the programs contained in the SPEC CPU 2006 benchmark [83]. The chapter also presents a comparison with the state-of-the-art heuristics. The goal of this preliminary analysis is twofold: first it can give insights on regularities and patterns that can be used in the subsequent branch predictor chapter, second it motivates the need for a better branch predictor. The analysis assesses the branch predictors in two ways: first the goodness of the predictions is evaluated with different metrics, and then the performance improvement achieved by employing the predictors is measured.

3.1 LLVM IR

LLVM's IR is the intermediate representation of the LLVM's compiler infrastructure. It exists in three form: as a human readable text file, as a bitcode binary file and as a data structure in LLVM [72]. The IR language is designed to match most of the high level constructs of programming languages while being easy to translate in one of the target architectures available. The human readable version is simple to understand. An example of the definition of a function that computes the Euclidean distance of a point (x, y) from the origin is the following:

```
define float @distance(float %x, float %y){
  entry_block:
  %mul = fmul float %x, %x
```

```
  %mul1 = fmul float %y, %y
  %add = fadd float %mul, %mul1
  %sqrtf = tail call float @sqrtf(float %add)
  ret float %sqrtf
}

declare float @sqrtf(float)
```

This snippet can help highlight some features of the IR. The language supports variables, functions, and types. Local variables are introduced by the `%` sign, while global symbols (such as function names) by `@`. The IR supports simple and structured types. The IR is by default in Static Single Assignment (SSA) form [1], which means that each variable is assigned only once. SSA is convenient for many analysis and optimizations. Basic blocks are referenced by labels like the `entry_block` label in the snippet above. Labels are regarded as local variables and are referenced by the control flow instructions.

LLVM's IR has only 3 types of control flow instructions: `br`(branch), `switch`, and `invoke` instructions. `br` instructions are of two types: conditional and unconditional. The latter is equivalent to a `goto`: the control flow is always directed to another basic block, while the former chooses one of the two directions to take upon evaluating a boolean variable. `switch` instructions can have more than two successors and `invoke` instructions are used to deal with runtime errors (similar to exception handling in programming languages).The different types of instructions are shown in the following snippet:

```
; unconditional branch
br label %next
; conditional branch
br i1 %condition, label %taken, label %nottaken
; switch
switch i32 %variable, label %default [
    i32 0, label %label1
    i32 1, label %label2
    i32 2, label %label3
]
; invoke
invoke void @function(...)
        to label %normal unwind label %exception
```

The unconditional branch is of no interest for this research as no control flow decision is to be predicted. Moreover both the `switch` and `invoke` instruc-

tions are not taken into account: the former because is infrequent and the latter because it is highly predictable as exceptional behaviour is supposed to be unlikely. The rest of the thesis focuses only on conditional branches.

3.2 Data

This Section describes the data used both in this chapter and in the next ones. Before describing the actual dataset used, let us make some general considerations.

The data we wish to use is an IR representation of a program with the probabilities associated to each branch instruction. Branch probabilities are obtained by profiling a program when running on some inputs, and then map the branch probabilities to branch instructions in the IR. This process highlights the fact that the data we are dealing with is composed of two interacting parts: the program, in the form of source code, and its inputs. The inputs affects the control flow and consequently the branch probabilities. To highlight this fact we can use an example: given a program, suppose we take as inputs for profiling its unit tests. The tests are built to cover all possible cases in a program, especially corner cases and error situations to find possible bugs. Testing inputs are crafted to execute non-frequent code regions therefore not representing the "average" control flow of the program. We argue that the branch probabilities should represent the average behaviour of the program such that the optimization have effect on most execution of the program.
To formalize this, suppose $I \in \mathcal{I}$ the random variable representing an input in the set of all inputs $\mathcal{I}$, and $B \in \{1, 0\}$ a branching instruction where value 1 means taken, and 0 not-taken. The probability distribution $p(B|I)$ represents the conditional probability from we are sampling from. The objective is to get the marginal probability of the branch, removing the dependency from the input:

$$p(B = 1) = \sum_{I \in \mathcal{I}} p(B = 1|I)p(I) \tag{3.1}$$

Assume we can partition the inputs set $\mathcal{I}$ into two classes: *normal inputs* $\mathcal{I}_N$, expressing the expected behaviour of the program, and *exceptional inputs* $\mathcal{I}_E$, representing exceptional behaviour of the program. Moreover assume the former are more likely than the latter $p(I \in \mathcal{I}_N) \gg p(I \in \mathcal{I}_E)$ we can write the approximation:

Benchmarking suite	Used By	Description
SPEC CPU '92/2000 /'06	[4, 96, 90, 14, 80]	Benchmark for performance evaluation of the CPU.
BioBench	[89]	Bioinformatics algorithms.
ptr-dist	[89]	Pointer intensive benchmark.
MediaBench II	[89]	Algorithms on media formats.
Polybench	[89]	Various numerical algoithms.

Table 3.1: Collection of some of the benchmarks used in previous work.

$$p(B = 1) \approx \sum_{I \in \mathcal{I}_N} p(B = 1|I)p(I)$$

As testing all the possible normal inputs is impossible for most programs, a subset of them will we used as a sample for the whole class.

Finding programs is not difficult given the many open source projects available, the problem is in acquiring relevant inputs for them. Previous works used benchmarks to perform their experiments because they are: standard, equipped with inputs, and built with the intent to measure a given metric. For example the SPEC CPU [39] is built to stress CPU and memory, while MediaBench [47] focuses on the performance of media encoding and decoding programs. Table 3.1 summarizes some of the benchmarks used in previous work. Programs will have various flow behaviour, some will have a more regular one, for example matrix multiplication, while others a more erratic one, *e.g.* particle simulator.

For this project we use SPEC's CPU2006 for two reasons: first it is a collection of different types of program spanning various areas, and secondly it allows to compare with earlier work on branch prediction as shown in the second column of Table 3.1. Table 3.2 instead, describes the content of the benchmark. Fortran programs are not yet considered due to a lack of standard frontend. Each program in SPEC's CPU is provided with three set of inputs: `train`, `test` and `ref`. Each of them has a purpose in the evaluation of the benchmark. Some comments about the choice of this dataset are given later in Section 3.6 when threats to validity are discussed.

3.2.1 Preprocessing

Preprocessing transforms raw data to a form that is suitable for the machine learning task. For this project the goal for preprocessing is to extract LLVM

Name	Language	Description
CINT - Integer benchmarks		
400.perlbench	C	PERL Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: go
456.hmmer	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics: Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing
CFP - Floating point benchmarks		
410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics: Quantum Chromodynamics
434.zeusmp	Fortran	Physics/CFD
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics/General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology/Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C/Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C/Fortran	Weather Prediction
482.sphinx3	C	Speech recognition

Table 3.2: Content of the SPEC CPU2006 benchmark. The benchmarks in red are not used.

IR code with branch probability information. To do that each program in the benchmark is profiled using Clang [88], a C frontend for LLVM. Programs are compiled with `-fprofile-instr-generate` and `-coverage-mapping` options. These two options create a profiling report for each run of the program. The report contains two information: how many times a function is executed and the probability of each branch, switch, and invoke instructions. LLVM stores branch probabilities as `branch_weights`: given a basic block, each of its successors is given an integer weight. Dividing each weight by the sum of all weights gives the probability of executing that block.

Each program in the benchmark is profiled on all given inputs sets(`train, test, ref`). Then each program is recompiled with the profiling information and emitted as LLVM IR. A sample of the workflow is shown in the script below.

```
clang -fprofile-instr-generate -fcoverage-mapping
 ↪  file.c -o file.o

# execute program

clang -fprofile-instr-use=file.profdata file.c
 ↪  -emit-llvm -S file.ll
```

The output is an annotated IR, such as the following one:

```
define i64 @factorial(i32) #0 !prof !28 {
  %2 = icmp sgt i32 %0, 1
  br i1 %2, label %3, label %11 !prof !29

; <label>:3:                ; preds = %1
  %4 = sext i32 %0 to i64
  br label %5

; <label>:5:                ; preds = %3, %5
  %6 = phi i64 [ %4, %3 ], [ %9, %5 ]
  %7 = phi i64 [ 1, %3 ], [ %8, %5 ]
  %8 = mul nsw i64 %7, %6
  %9 = add nsw i64 %6, -1
  %10 = icmp sgt i64 %6, 2
  br i1 %10, label %5, label %11 !prof !30
```

```
; <label>:11:                ; preds = %5, %1
  %12 = phi i64 [ 1, %1 ], [ %8, %5 ]
  ret i64 %12
}

...

!28 = !{!"function_entry_count", i64 0}
!29 = !{!"branch_weights", i32 9, i32 1}
!30 = !{!"branch_weights", i32 3, i32 7}
```

The IR contains the profiling information in the form of metadata annotations. The profiling metadata is described by the `!prof` token, followed by a metadata entry number such as `!30`. The meaning of each metadata entry is listed at the end of the file. Referring back to the previous example, `!30` represents branch weights, leading to the probability $\frac{3}{3+7}$ and $\frac{7}{3+7}$.

3.3 Evaluating LLVM's Predictor

The first experiment conducted is analyzing LLVM's branch predictor accuracy. As already mentioned in subsection 2.3.3, LLVM uses a set of heuristics to predict branch probabilities. Each heuristic assigns the same probability to the branches for which it applies. As a result there is a finite set of probability that can be assigned to branches. These are listed in Table 3.3.

3.3.1 Comparison with Heuristics

As cited in Subsection 2.3.3, LLVM's predictor implements a subset of the heuristic defined by Ball and Larus [4] and add some language specific ones. To the best of the author's knowledge the reason why the developer did not implement all Ball and Larus heuristics is not known nor mentioned anywhere. For completeness of comparison we implemented Ball and Larus [4] and Wu and Larus [96] heuristics, as they are the only ones that assigns probability to branches among the related work in Section 2.3.

We collected the branches for which there was profiling information. For each of them the profiled branch probability is retrieved (the ground truth). For each of these branches we also collected the predicted branch probability, one for each heuristic. We restrict the focus only on the taken direction of each branch, as the non taken is symmetric and will lead to the same result.

Taken	**Not-taken**	**Description**
96.8%	3.2%	Loop heuristic.
5.8%	94.2%	Cold call heuristic
62.5%	37.5%	Pointer, Zero, Floating point heuristics
$\sim 100\%$	$\sim 0\%$	Invoke heuristic
50%	50%	If none of above applies

Table 3.3: Possible branch probabilities assigned by LLVM's branch heuristic mechanism.

For a branch b call θ_b the predicted branch probability and $\hat{\theta}_b$ the ground truth branch probability. Let's define the indicator function $I(x = y)$ as follows:

$$I(x = y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases} \tag{3.2}$$

Moreover for a branch b we define its execution count ec_b the number of times it has been executed during the profiling, and t_b the number of times the branch has gone the taken direction. Symmetrically $\bar{t}_b$ is the number of times the not-taken direction is followed. Of course $ec_b = \bar{t}_b + t_b$ must holds for each branch b. The heuristic are evaluated and compared taking two different perspective: as a branch predictor and as a branch probability predictor.

As a branch predictor Branch predictors are binary classifiers, they predict what will be the most likely taken successor of a branch. For convention assume a prediction of 1 to represent the taken successor to be the most likely and 0 for the opposite.
A heuristic that assigns a probability θ to a branch, can be turned into a branch predictor with the following function:

$$h(\theta) = \begin{cases} 1 & \text{if } \theta \geq 0.5 \\ 0 & \text{if } \theta < 0.5 \end{cases} \tag{3.3}$$

Turning each heuristic and the profiled branch probability into branch predictors it is possible to compute two metrics: accuracy and miss rate. These two metrics are used in most of the previous researches in branch prediction of Section 2.3.

Accuracy measures the percentage of branches correctly predicted with the following formula:

$$accuracy = \frac{\sum_{b \in \mathcal{B}} I(h(\theta_b) = h(\hat{\theta}_b))}{|\mathcal{B}|} \tag{3.4}$$

This metric is important, but as remarked in the introduction, the main target for optimization are branches executed more frequently. Therefore a more insightful metric is the miss rate, which is the percentage of the execution of branches that are mispredicted. The miss rate takes into account the execution count of each branch.

$$miss\_rate = \frac{\sum_{b \in \mathcal{B}} t_b \cdot I(h(\theta_b) = 0) + \bar{t}_b \cdot I(h(\theta_b) = 1)}{\sum_{b \in \mathcal{B}} ec_b} \tag{3.5}$$

The miss rate is lowerbounded by the Perfect Branch Predictor (PBP), which is the miss rate obtained by using the branch predictor derived from the ground truth branch probabilities ($\hat{\theta}_b$ instead of θ_b).

As a probability branch predictor Branch probability predictors assigns to each branch a value in the interval $[0, 1]$. As a probability branch predictor we want to measure the discrepancy between the predicted branch probability and the true one. A straightforward metric is the mean squared error (MSE):

$$MSE = \frac{\sum_{b \in \mathcal{B}} (\theta_b - \theta_b)^2}{|\mathcal{B}|} \tag{3.6}$$

The weighted mean squared error (WMSE) is also computed to include the execution count of each branch.

$$WMSE = \frac{\sum_{b \in \mathcal{B}} ec_b \cdot (\theta_b - \theta_b)^2}{\sum_{b \in \mathcal{B}} ec_b} \tag{3.7}$$

The above metrics are computed on SPEC CPU 2006 benchmark. They are firstly evaluated on each programs of Table 3.2 separately and then on the benchmark as a whole. The results are presented later in Section 3.5.

3.4 Branch Probability effect on Performance

The goal of this experiment is understanding what is the effect of the branch predictor on the performance of the program. Before explaining the evaluation method, the two following paragraphs explain the effect of branch probabilities

in LLVM's optimization. The effect of function counts is also explained as it is part of the profiling information used later in the evaluation.

Passes affected by branch probabilities

The branch prediction is invoked by using the `-branch-prob` optimization pass. This pass is run many times in the default optimizations levels `-O1`, `-O2`, and `-O3`. Branch probabilities are used directly and indirectly by many components of LLVM. Their main use is in code generation as they drive the block placement in the binary file. Other uses can be found in Transforms (IR to IR transformations) such as: `PartialInlining`, `GuardWidening`, and `JumpThreading` where the branch probabilities are used as a measure for the usefulness of the transformation. Finally branch probabilities are used widely in target-specific operations.

As describe in Section 2.1, branch probabilities are the basis for computing basic block frequencies, in fact LLVM's optimization levels pairs the `branch-prob` pass with the `block-freq` pass which approximate the solution obtainable with equation 2.2. Therefore all the passes influenced by block frequencies are affected by the branch prediction mechanism. Block frequencies are used in register allocation, function inlining, and spilling [84]. Block frequencies are used as a decision variable for applying or not loop transformation such as : `Loop Unrolling`, and `Loop Hoisting`.

Passes affected by function counts

Function count is the other information collected with profiling along with the branch probabilities. Function counts are only used in function inlining. Contrary to the case of branch probabilities, LLVM treats differently the case where function counts are coming from profiling and the estimated ones.

Evaluation

The evaluation compares the running time of the benchmark compiled with the heuristics used in the previous sections. Two other configurations derived from each program's profile are also tested: one is compiling with the full profiling information, and the other with profiling information deprived with the function counts, as summarized in Table 3.4. These two serve as reference as having the "perfect" information of each program's behaviour.

Each configuration is compiled with the `-O2` optimization level. This is because we want to see if the branch probabilities have a performance impact at

Configuration	Branch Data	function Count
Only Branch	✓	✗
Profile	✓	✓

Table 3.4: The two additional compilation configurations that use profiled information.

the maximum optimization level. It is also relevant from a user perspective, as it usually it interfaces with the default optimization levels. The benchmarks are run on a computer equipped with an Intel i7-4710HQ processor and DDR3-1600 RAM. The benchmarks are run 3 times, and the median time for each benchmark is reported. This is the default method of evaluating SPEC CPU benchmarks.

The speedup is computed between each of the configuration and the basic LLVM predictor according to the following equation:

$$Speedup = \begin{cases} \frac{T_{LLVM} - T_{opt}}{T_{LLVM}}, & \text{if } T_{LLVM} < T_{opt} \\ \frac{T_{LLVM} - T_{opt}}{T_{opt}}, & \text{if } T_{LLVM} > T_{opt} \end{cases} \tag{3.8}$$

Where T_{opt} is the execution time of either one of the heuristic, while T_{LLVM} represents the execution time of the basic LLVM branch predictor. If $T_{LLVM} < T_{opt}$ the speedup is negative and represents a slowdown. The Equation 3.8 is symmetric and therefore speedups and slowdowns can be compared. Moreover, to see the branch probability impact on function inlining, we also repeat the evaluation with Link Time Optimization (LTO) enabled. LTO enable inlining between compilation unit which uses branch probabilities. The speedup of the LTO case is computed with respect to the basic LLVM predictor compiled with LTO enabled.

3.5 Results

From the profile information we obtain the distribution of branch probability for the taken direction, as shown in the left plot of Figure 3.1. We can further analyze the distribution by taking the probability of the least frequent (symmetrically the most frequent) direction for each branch. The distribution is shown on the right plot of Figure 3.1. By taking the cumulative of this distribution one can infer that roughly 80% of the branches take one of the two directions with more than 80% probability. This confirms the finding of Fisher and Freudenberg [29] on the predictability of branches.

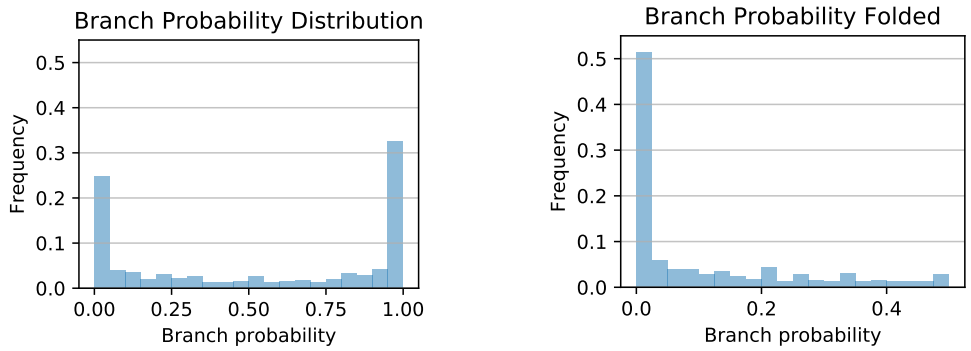


Figure 3.1: Branch probability distribution on SPEC's CPU2006 benchmark. On the right the distribution represents the taken branch probability. On the right the distribution has been "folded" by using the symmetry at 0.5.

In Figure 3.2 the distribution of the taken probability is overlapped with the distribution of branch probabilities predicted by LLVM. The discretization of the frequencies is due to LLVM's predictor itself as described in subsection 3.3. It is immediately evident that the two distribution differ, notably LLVM's predictor assigns more probability mass around 0.5 while the profilied probabilities are more dense around the extremes of the $[0, 1]$ interval. Moreover, LLVM's heuristic fail to assign branch probabilities almost half of the times. The picture can be misleading as it does not take into account the number of times a certain branch is executed.

The miss rate and accuracy metric are shown in Table 3.5 while MSE and WMSE in Table 3.6. The different heuristics are referred as follows: "LLVM" is LLVM's predictor, "BALL" is the Ball and Larus heuristic, and "WU" is the Wu and Larus heuristic. The perfect branch predictor is referred with the "PBP" label.

Both Ball and Larus and Wu and Larus heuristics have a better accuracy and miss rate than the basic LLVM branch predictor, while for MSE and WMSE LLVM's branch predictor outperforms the other two heuristics.

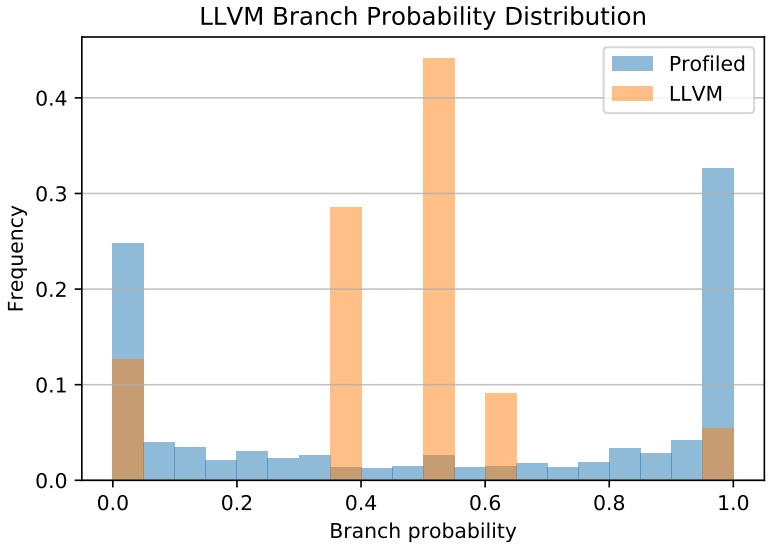


Figure 3.2: Branch probability distribution on SPEC's CPU2006 benchmark, LLVM's prediction is compared with the profiled distribution.

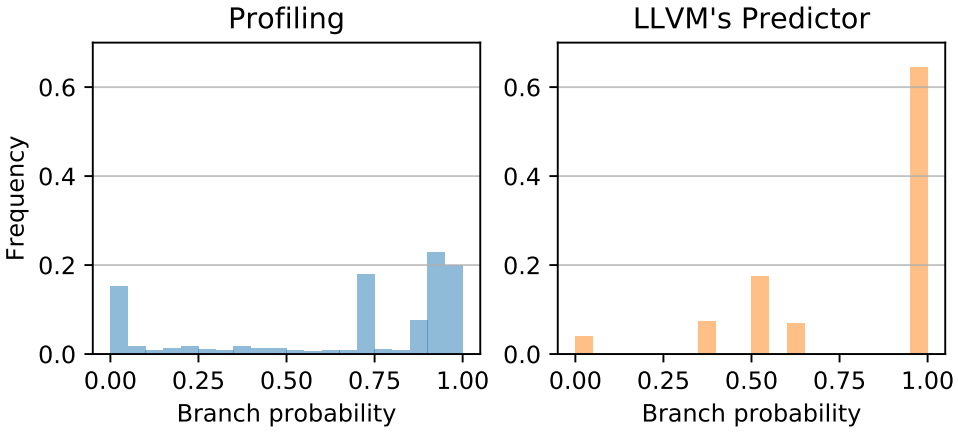


Figure 3.3: Distribution of branch probabilities weighted by the number of time the branch is executed.

Benchmarks	Accuracy			Miss Rate			
	LLVM	BALL	WU	LLVM	BALL	WU	PBP
400.perlbench	49.845%	**57.714%**	56.284%	28.093%	**17.906%**	20.544%	5.199%
401.bzip2	67.077%	**76.572%**	75.215%	23.665%	**15.977%**	28.833%	9.717%
403.gcc	50.009%	**58.203%**	57.278%	37.502%	**28.580%**	31.807%	9.738%
429.mcf	**69.277%**	68.674%	66.867%	**28.145%**	38.827%	29.590%	13.750%
445.gobmk	53.005%	**55.935%**	55.213%	41.832%	**36.482%**	39.890%	16.830%
456.hmmer	61.704%	**70.356%**	63.613%	62.794%	**19.659%**	19.787%	9.161%
458.sjeng	51.203%	56.541%	**58.646%**	38.079%	30.862%	**29.458%**	15.531%
462.libquantum	69.406%	**75.799%**	75.342%	34.673%	**34.667%**	34.679%	11.528%
464.h264ref	55.006%	**61.670%**	56.887%	33.521%	**24.902%**	28.907%	14.363%
471.omnetpp	**70.120%**	69.609%	68.912%	**39.542%**	41.541%	45.542%	11.162%
473.astar	66.842%	**72.631%**	67.105%	40.598%	**28.758%**	33.393%	22.267%
483.xalancbmk	**60.542%**	58.312%	56.034%	**10.311%**	15.056%	10.610%	8.691%
433.milc	69.230%	**71.400%**	64.497%	22.696%	**22.651%**	23.301%	22.602%
444.namd	43.934%	**49.398%**	39.344%	43.624%	**20.430%**	62.409%	11.169%
447.dealII	**70.730%**	60.070%	56.769%	**22.515%**	24.756%	26.372%	21.811%
450.soplex	66.192%	**69.293%**	65.582%	15.169%	**10.385%**	15.666%	7.701%
453.povray	46.468%	**56.270%**	52.475%	55.075%	**49.330%**	52.361%	14.799%
470.lbm	63.157%	**84.210%**	77.192%	61.641%	**30.449%**	30.502%	4.289%
482.sphinx3	63.176%	**76.444%**	71.570%	29.426%	**21.746%**	21.991%	5.668%
Total	54.245%	**59.137%**	57.138%	22.924 %	21.122%	**20.798%**	12.508 %

Table 3.5: Accuracy and Miss Rate for the different branch probability predictors.

Benchmarks	Mean Squared Error			Weighted Mean Squared Error		
	LLVM	BALL	WU	LLVM	BALL	WU
400.perlbench	**0.187**	0.187	0.197	0.111	**0.110**	0.171
401.bzip2	0.148	**0.126**	0.139	0.067	**0.067**	0.097
403.gcc	**0.190**	0.195	0.205	0.121	**0.120**	0.141
429.mcf	**0.125**	0.156	0.154	**0.114**	0.176	0.131
445.gobmk	**0.151**	0.180	0.191	0.128	**0.121**	0.134
456.hmmer	**0.129**	0.139	0.168	0.137	**0.124**	0.126
458.sjeng	**0.154**	0.182	0.185	**0.088**	0.101	0.098
462.libquantum	**0.082**	0.124	0.129	**0.068**	0.184	0.183
464.h264ref	**0.154**	0.181	0.193	**0.062**	0.078	0.089
471.omnetpp	**0.128**	0.160	0.160	**0.155**	0.195	0.190
473.astar	**0.117**	0.132	0.163	0.070	**0.061**	0.069
483.xalancbmk	**0.161**	0.193	0.210	0.025	0.065	**0.022**
433.milc	**0.091**	0.110	0.159	0.042	**0.006**	0.020
444.namd	**0.227**	0.260	0.281	**0.079**	0.083	0.180
447.dealII	**0.150**	0.193	0.205	0.060	0.031	**0.030**
450.soplex	**0.135**	0.160	0.164	**0.060**	0.066	0.078
453.povray	**0.175**	0.193	0.231	0.216	**0.200**	0.243
470.lbm	0.114	**0.104**	0.119	0.126	**0.124**	0.125
482.sphinx3	0.123	**0.121**	0.132	**0.079**	0.135	0.153
Total	**0.173**	0.187	0.199	**0.059**	0.076	0.064

Table 3.6: Mean Squared Error and Weighted Mean Squared Error for the different branch probability predictors.

3.5.1 Performance Improvement

The running times for each benchmark and heuristic are reported in Appendix B. The Speedups for the benchmarks are represented in Figure 3.4 for the case without LTO, and Figure 3.5 for the LTO case. Both pictures represents the average speedups for each configuration.

For both with and without LTO, the average speedups of all the configuration tested is positive. Moreover the average of Ball and Larus and Wu and Larus heuristics coincide in both cases. Slowdowns occur rarely among the benchmarks. A significant divide can be observed between the profiled program and the heuristics only in the LTO case.

3.6 Conclusions

The previous analysis shows that the state-of-the-art heuristics outperform LLVM's branch probability prediction in terms of accuracy and miss rate. Moreover both Ball and Larus and Wu and Larus heuristics cause a positive speedup on almost every program in the benchmark both with and without LTO. Assuming the speedup can be replicated for other programs than the one considered, the implementations of the Ball and Larus and Wu and Larus heuristics are an important contribution to LLVM's framework considering its widespread use.

Table 3.5 shows that there is still room for improvement in the miss rate and accuracy of the branch predictor. The speedup obtained by improving the prediction is more difficult to interpret: the use of profile information (whether only branch probabilities or the full profile) proves consistently better than the heuristics when compiled with LTO. This fits with the assumption that better branch predictors gives better results, hence using profiling information gives an upperbound on the speedup. Although, when the programs are compiled without LTO this assumption does not hold anymore: for some programs the heuristics outperform the profiled information, and the average speedup of the heuristics and the profiled one are almost identical.

Within LLVM a more accurate branch predictor will most likely show its benefit when a program is compiled with LTO. Moreover more aggressive compilers such as UNISON [50, 49] could benefit from more accurate branch probabilities.

Another comment regarding the experimental setup has to be addressed: the speedups are dependent on the hardware the program runs on. The speedup results are obtained on a CPU with a CISC architecture [37], which includes

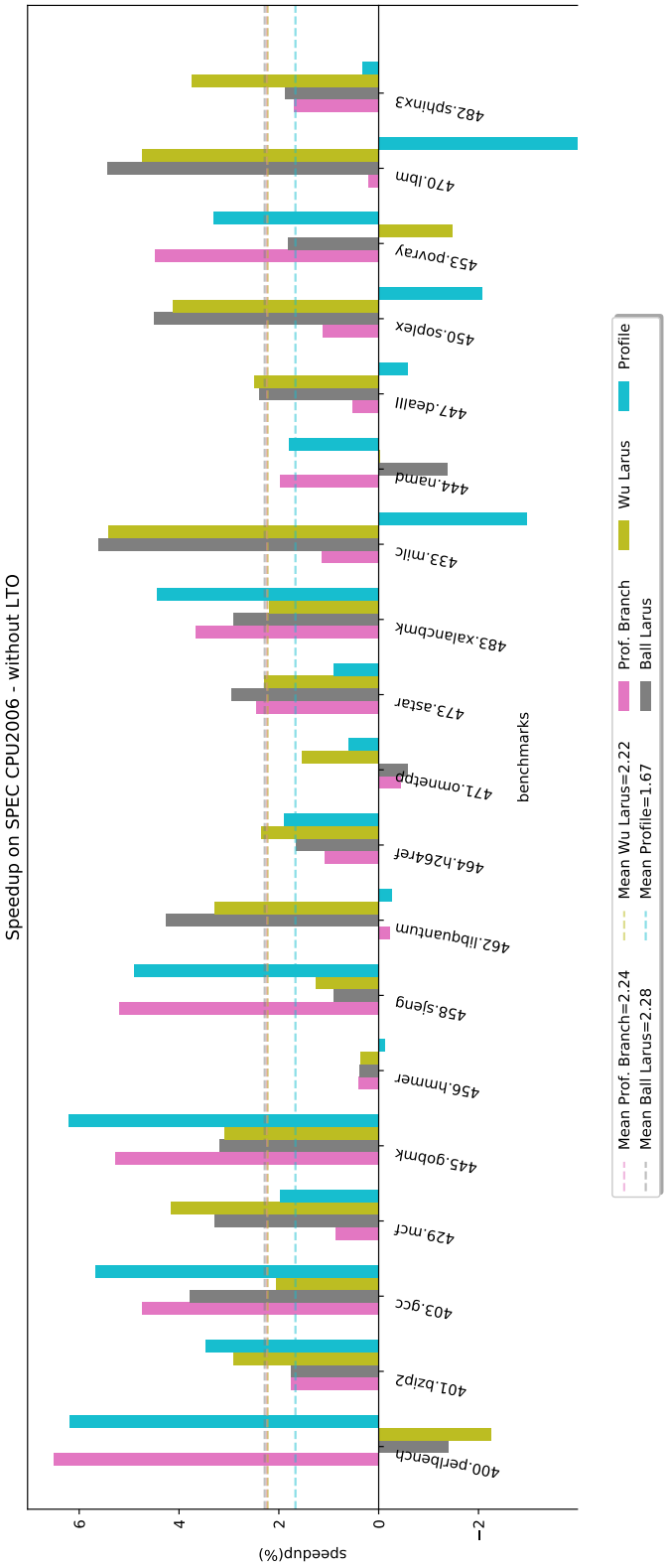


Figure 3.4: Speedup on SPEC CPU 2006. Speedups obtained on LLVM's branch predictor as described in Section 3.4 without LTO. The means of Ball and Larus, Wu and Larus and Prof. Branch configurations are overlapped.

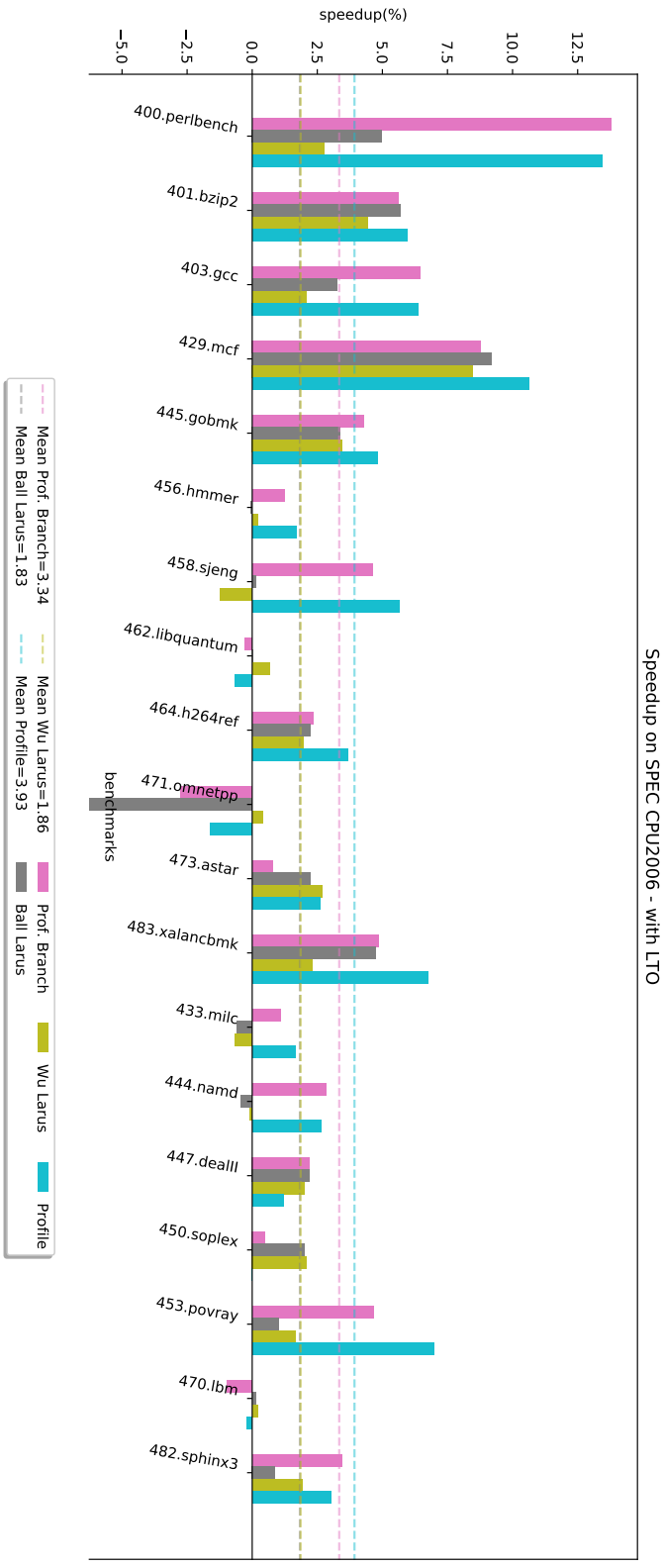


Figure 3.5: Speedup on SPEC CPU 2006. Speedups obtained on LLVM's branch predictor as described in Section 3.4 with LTO enabled. The means of Ball and Larus and Wu and Larus configurations are overlapped.

hardware optimization that can mitigate the effect of a poor branch predictor. We can only speculate to obtain different speedup results (but not necessarily better) on a RISC architecture, which are more simple and reliant on compiler clues.

Threats to Validity Taking this benchmark as a representation of programs still require a motivation. We do not have a more convincing argument than the availability of the data, but certainly the benchmark was not designed to represent a wide variety of programs behaviour. As stated in the documentation [38]: *"SPEC CPU2006 focuses on compute intensive performance, which means these benchmarks emphasize the performance of: the computer processor (CPU), the memory architecture, and the compilers "*. Some work has been done in analyzing redundancy in SPEC CPU itself [65] concluding that some programs share similar behaviour (considering branches, memory accesses and cache access). Many other real-life programs may not benefit so much of a better prediction of probabilities, *e.g.* a web server may be bottlenecked by the network's latency, or a rendering program might be limited by the disk's writing and reading speed.

Considering the `-O2` optimization level as the baseline for the evaluation may be incorrect as the optimization process is just a hand-crafted sequence of passes and algorithms carefully tuned. The snippet of Listing 1, taken from LLVM's inlining pass, shows how the optimization process is influenced by hard coded parameters, and therefore can induce a speedup or a slowdown. Because of this we are not assured that the performance is monotonic increasing as the prediction of the branches gets better.

```
// Various magic constants used to adjust
 ↪  heuristics.
const int InstrCost = 5;
const int IndirectCallThreshold = 100;
const int CallPenalty = 25;
const int LastCallToStaticBonus = 15000;
const int ColdccPenalty = 2000;
```

Listing 1: Small excerpt from LLVM's `InlineCost` class(`https://llvm.org/doxygen/InlineCost_8h_source.html#l00042`).

Chapter 4

Representation Learning Branch Predictor

This Chapter describes the methodologies used to address the main question of the thesis: predicting branch probabilities through representation learning. Section 4.2 describes the preprocessing of the data and the embedding of the instructions. Section 4.3 describes the model used to predict the branch probabilities. The evaluation of the model is left to Chapter 5.

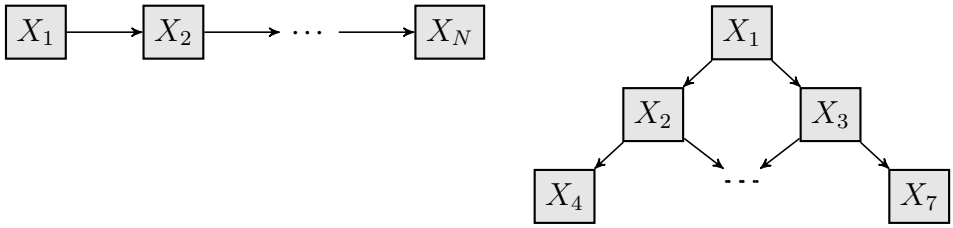
4.1 Solution Outline and Intuition

Each instruction in a programming language represents an operation and has a precise formal semantic, that is the relation between its inputs and its outputs. The semantic of groups of instruction can represent an abstract operation such as *"add element to a hash table"* or *"sort a list"*. These operation can emerge as patterns coming directly from the programmer's implementation of a function. Note that the semantic of the program comes from the interplay of two factors: the instruction performed, and the data structure used [22, 94]. Just like humans can have an intuition about the semantic and behaviour of a program solely by reading it, we assume under the *naturalness hypothesis* (see Section 2.5) that by using natural language processing it is possible to predict branch probabilities by "reading" a program.

A practical realization of this intuition is given by the following example: consider two program snippets, one using a linked list and the other using a balanced binary tree as in Figure 4.1. Suppose two algorithms traverse each data structure: to test the stopping condition both check the pointer(s) for the next element(s) in the data structure. For the linked list it checks the pointer to

the next element to be NULL, while for the tree it checks for each of the two children's pointer to be NULL. That test has a different probability of being true depending on the data structure used. In the example for the linked list only the last element will make the condition true, so the branch probability is $\frac{1}{N}$. For the binary tree every leaf has a NULL pointer and therefore the probability $\frac{\lceil N/2 \rceil}{N}$. As the number of elements increases the first tends to 0 while the other tends to $\frac{1}{2}$. By leveraging the information about the data structure used, it is possible to draw conclusion about the branch probabilities. The heuristics cited in Subsection 2.3.1 would not have distinguished between the two data structures because they are designed for assembly languages which does not have such information.

LLVM's IR types represents the information about the data structure used. Our hypothesis is that by leveraging semantic information contained in the IR we can obtain accurate prediction of its branching behaviour. The amount of possible data structures and the different combinations of them creates a massive amount of cases to cover by heuristics. This motivates the use of representation learning algorithm to automatically deal with the problem.




```
struct list {
    void* content;
    struct list* next;
}

// traversal algorithm
if(list->next == NULL)
```

```
struct tree {
    void* content;
    struct tree* left;
    struct tree* right;
}

// traversal algorithm
if(tree->left == NULL)
```

Figure 4.1: Example of how semantic information can help in predicting branches.

4.2 Preprocessing

This section describes the extraction of the data used for training and testing the branch predictor. The source dataset is SPEC CPU 2006, the same described and used in Section 3.2. Before describing the the preprocessing step, a brief observation is required; for any machine learning application data is the most critical part. It is critical for two reasons: first it represents the examples to learn from, secondly the representativity of the data is what makes a learned model general, applicable to other datasets. For these reasons the optimal dataset must be rich with different types of branches and must represent many types of program types and behaviours. We have no guarantee about the representativity of the dataset in use. Some work [65] have analyzed the similarities inside the benchmark itself identifying some redundancy, but have not compared with other programs behaviour.

The starting point of the preprocessing is the LLVM IR resulting from the compilation of each compilation unit in the benchmarks. On the IR the binary branches for which we have profiling information (the ground truth to learn from) are identified. The outcome of a branch is determined by the instructions executed before, but for prediction purposes the instructions executed in the target blocks might give useful information (*e.g.* trigger an exception, exit the function, etc.). The information about the target blocks is also used in the most of the heuristics explored in Subsection 2.3.1 [14, 4, 96]. Therefore the instructions of the 3 basic blocks involved in a branch are extracted from the IR and all of them are used for predicting the branch probability as depicted by Figure 4.2.

$$p = p(BB_1 \rightarrow BB_2 | BB_1, BB_2, BB_3)$$

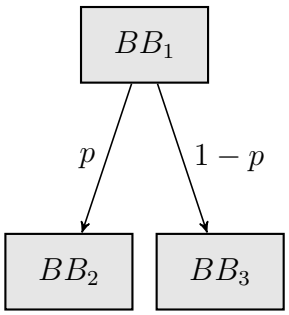


Figure 4.2: The goal of the predictor is to predict the branch probability p between basic blocks BB_1 and BB_2 given the instruction of BB_1, BB_2 and BB_3.

It is possible that the basic blocks contain only a small amount of instruc-

tion. For this reason we include instructions from other basic block with the following rule: if BB_1 has only one predecessor, it too is included as the list of instructions before the branch. A similar procedure applies for $BB2$ and $BB3$: if any of them has only one successor, it too is appended to the list of instructions. An example of how this rule works is shown in Figure 4.3. The number of collected instructions for each basic block is limited to 50 to avoid using unnecessary instructions.

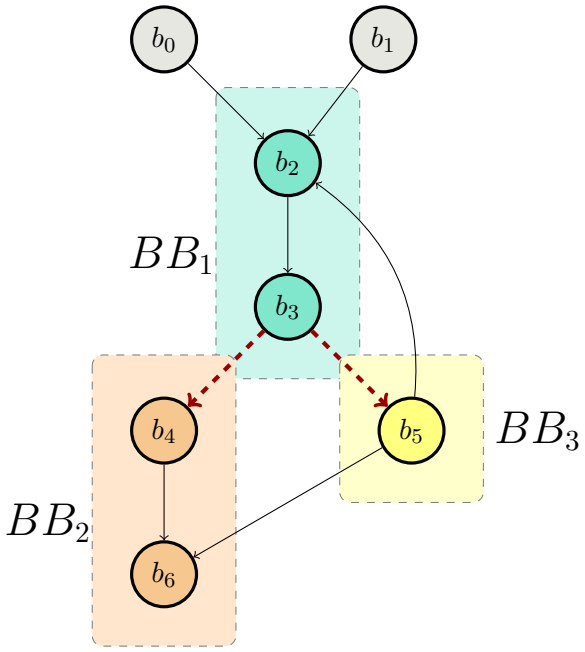


Figure 4.3: Extraction of the instruction from the CFG. Consider the branch in b_3 and the successors b_4 and b_5. Each colored rectangle represents the the basic blocks from which the instructions are taken.

Each list of instructions has to be further processed to be easily used in any machine learning algorithm which rarely use the textual form of the input. The next subsection presents how the encoding of the instructions is performed using `inst2vec` [6], a model to learn embeddings for LLVM's IR instructions.

4.2.1 Embedding and `inst2vec`

`inst2vec` is an algorithm for learning embeddings of LLVM's IR. `inst2vec` builds from `word2vec` described in Section 2.6. `inst2vec` does two things: normalizes the instruction to avoid a huge vocabulary, and it redefines the concept of context ($Ctx(w)$) of a word (instruction) w.

The normalization of the instructions is described later, let's start with the definition of the context. `inst2vec` defines the *context* informally as *"statements whose execution directly depends on each other"* [6]. The formal definition of the context is built from two binary relations over the instructions: *execution dependence* and *data dependence*. An instruction l is execution dependence from w if it can be executed immediately after w. An instruction l is data dependent from w if it uses a value defined or modified by instruction w. Given an integer C called *context size*, define the context of a word w as the set of instructions found by recursively applying the definitions of data and execution dependence no more than C times. More information can be found in the original `inst2vec` paper [6].

The vocabulary is built by first processing the instructions to remove all the sources of variations in the instructions that would increase the vocabulary size greatly. First the IR is stripped out of the unnecessary elements: comments, metadata, attributes, declared functions, target architecture, source filename, call to assembly functions, assembly modules, COMDATS (which specify executable file properties). Statements on multiple lines are aligned into one. Structures are also inlined by substituting the named structures with its definition recursively until there are no more named structures. Then identifiers (such as variable names) are substitute by a token `<ID>` while immediates are substituted by a token specifying its type, such as `<INT>`, `<FLOAT>`. Immediates are saved separately to be used as an auxiliary input.

The vocabulary is reduced by removing those instructions which appear less than 50 times in the sources. This greatly reduces the size of the vocabulary. Instructions that are not present in the vocabulary are substituted with an `<UKN>` token. An example is given by considering a subset of the instructions of Listing 2, they are preprocessed in Listing 3. The embedding are learned by using the Skip-gram model [58] described in 2.6.

```
struct node{
    int value;
    struct node** nexts;
    int visited;
};

void mark_visted(struct node* vertex){
    if(vertex->visited){
        return;
    }else{
        vertex->visited = 1;
    }
}
```

```
%struct.node = type { i32, %struct.node**, i32 }

define void @mark_visted(%struct.node*) #0 {
  %2 = alloca %struct.node*, align 8
  store %struct.node* %0, %struct.node** %2, align 8
  %3 = load %struct.node*, %struct.node** %2, align 8
  %4 = getelementptr inbounds %struct.node, %struct.node* %3,
  ↪  i32 0, i32 2
  %5 = load i32, i32* %4, align 8
  %6 = icmp ne i32 %5, 0
  br i1 %6, label %7, label %8
; <label>:7:        ; preds = %1
  br label %11
; <label>:8:        ; preds = %1
  %9 = load %struct.node*, %struct.node** %2, align 8
  %10 = getelementptr inbounds %struct.node, %struct.node*
  ↪  %9, i32 0, i32 2
  store i32 1, i32* %10, align 8
  br label %11
; <label>:11:       ; preds = %7, %8
  ret void
}
```

Listing 2: Correspondence between a C function and its LLVM's IR equivalent representation.

```
define void @mark_visted(%struct.node*)
define void <@ID>({ i32, opaque**, i32 }*)

%2 = alloca %struct.node*, align 8
<%ID> = alloca { i32, opaque**, i32 }*, align 8

store %struct.node* %0, %struct.node** %2, align 8
store { i32, opaque**, i32 }* <%ID>, { i32, opaque**,
 ↪  i32 }** <%ID>, align 8

%3 = load %struct.node*, %struct.node** %2, align 8
<%ID> = load { i32, opaque**, i32 }*, { i32, opaque**,
 ↪  i32 }** <%ID>, align 8

%5 = load i32, i32* %4, align 8
<%ID> = load i32, i32* <%ID>, align 8

%6 = icmp ne i32 %5, 0
<%ID> = icmp ne i32 <%ID>, <%INT>

br i1 %6, label %7, label %8
br i1 <%ID>, label <%ID>, label <%ID>
```

Listing 3: Some examples of the preprocessing step on a subset of instructions of Listing 2. The original instructions are highlighted in light blue.

4.3 Model

The following section describes the architecture and the design decisions of the branch probability predictor. We remark that contrary to all previous approaches, in this project the encoding uses an unsupervised embedding algorithm to encode the input. The training and evaluation methods are also described. The results are reported in the next chapter.

4.3.1 Predictor architecture

Figure 4.4 shows the predictor's high level architecture. Given a binary branch in LLVM's IR, the three basic block involved are extracted as described in Section 4.2, and each of the instruction is encoded using `inst2vec`. Each basic block's embedded instructions are fed into a a type of RNN called Long Short Term Memory (LSTM) network [40]. LSTMs are covered in Appendix A but the basic structure shown in Figure 2.5 still holds for LSTMs. The use of LSTM over other RNN architecture is motivated by their empirical reliability and common use in other NLP tasks [82, 55, 87, 92]. At the output of each LSTM layer we obtain a single vector, one for each basic block. In an analogy with NLP, if instructions are like words, basic blocks are like paragraphs.

The three representation of the basic blocks are then fed into a Feed Forward Neural Network (as described in Subsection 2.4.1). The network has an additional input that describes the loop structure of the branch, as both Fisher and Freudenberger [29] and Ball [4] remarked their highly importance and easy to predict. The additional information represents the cases where the branch: is a backedge (`B`) of the loop, exits a loop (`EX`), or none of the previous (`DK`). The output of the FFNN is the predicted probability of the branch to be taken.

Parameters The embedding size of `inst2vec` is chosen to be 128. The output size of the LSTM cells is of size 128, the same size as the input vectors. The LSTM cells have no particular optimization as suggested by Greff *et al.* [33]. The FFNN has two layers, one of size 256 and the other of 64. The activation functions in the hidden layer are $tanh$ (to avoid vanishing gradient) and the output activation function is a sigmoid to obtain an output in the range $[0, 1]$.

Loss Function One can derive a loss function by computing the likelihood of the data. To do that one has to model the outcomes of the branch statistically.

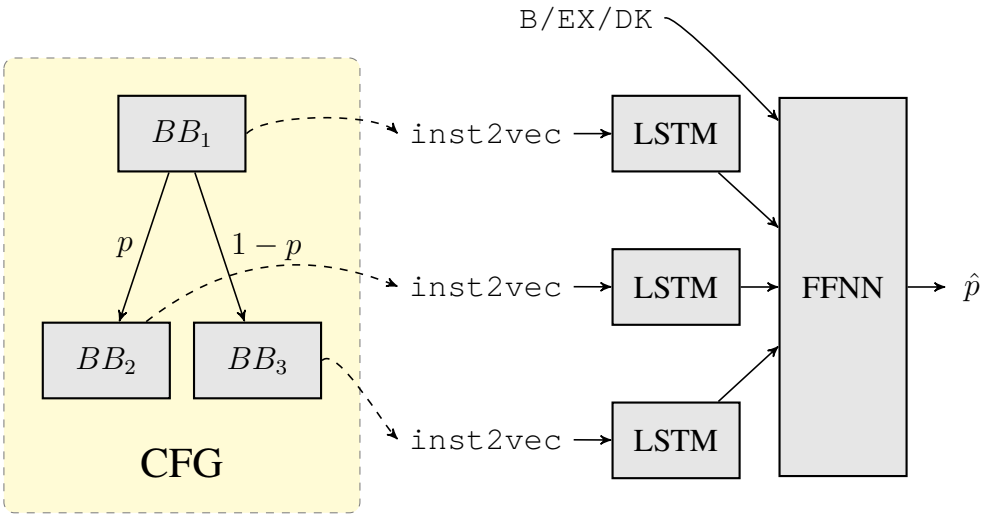


Figure 4.4: High level architecture for the branch predictor proposed in the thesis.

Calder *et al.* [14] used a Bernoulli random variable to model each branch thus using a cross-entropy loss function. As they point out themselves, this will not predict to the probability of the branch itself. Therefore this will only increase the accuracy of the predictor and not the precision of the branch probabilities.

A branch execution can be modeled as a sequence of Bernoulli experiments (branch taken or not taken) therefore, assuming independence between the executions, a binomial random variable is suitable for describing multiple execution of the branch. Given a branch, calling ec its execution count, t the number of times the branch has been taken, and $\bar{t} = ec - t$ the number of times it is not taken, the likelihood of the predicted branch probability $\hat{\theta}$ is:

$$l(\hat{\theta}) = \binom{ec}{t} \left(\hat{\theta}\right)^t \left(1 - \hat{\theta}\right)^{ec-t} \tag{4.1}$$

Taking the negative logarithm of the above quantity we get the following loss function:

$$\mathcal{L}(\hat{\theta}) = -\,log\left(l(\hat{\theta})\right) = \tag{4.2}$$

$$= -\,log\left[\binom{ec}{t}\right] - t \cdot log\left(\hat{\theta}\right) - (ec - t) \cdot log\left(1 - \hat{\theta}\right) \tag{4.3}$$

Note that the above equation takes into account the execution count of each branch. The minimization of $\mathcal{L}(\hat{\theta})$ is hence equivalent to minimizing the miss

rate of the predictor.
For a single branch we can find the global minimum the parameter $\hat{\theta}$ will converge to. Taking the derivative we obtain the following:

$$\frac{\partial \mathcal{L}(\hat{\theta})}{\partial \hat{\theta}} = -t \cdot \frac{1}{\hat{\theta}} + (ec - t) \cdot \frac{1}{1 - \hat{\theta}} \tag{4.4}$$

Solving for the derivative to be 0 we get:

$$\frac{\partial \mathcal{L}(\hat{\theta})}{\partial \hat{\theta}} = 0 \tag{4.5}$$

$$\implies \hat{\theta} = \frac{t}{ec} \tag{4.6}$$

The optimal branch probability tends to the fraction of the execution where the taken direction is executed. The derivative of Equation 4.4 is used in gradient descent to train the predictor. Given the magnitude of the execution count of some branches (in the order of 10^9 or above), there is the risk of numerical problems in the computation of the derivative or, more importantly, exploding gradient[32] hence a more difficult convergence to the minima of the function. Given these possible problem the mean squared error loss between the predicted branch probability $\hat{\theta}$ and the true one $\theta = \frac{t}{ec}$ for each branch b_i.

$$\mathcal{L}(\theta, \hat{\theta}) = \frac{1}{K} \sum_{i=1}^{K} \left(\theta_{b_i} - \hat{\theta_{b_i}} \right)^2 \tag{4.7}$$

Where K is the batch size. The above loss aims at minimizing the error between the predicted and true branch probability, regardless of the number of times a branch is executed.

4.3.2 Evaluation Methods

Training Training is done into two steps: first `inst2vec` is trained on all benchmark's sources, then the classifier is trained using the obtained embedding. During the predictor's training the embedding are not changed. `inst-2vec` is trained for 5 epochs with a context size of 2, the same as described in its original paper [6].

Training Parameters The learning rate is controlled by the ADAM [44] algorithm. The division of training and testing set is done on a benchmark level so that the model learns program-independent features.

Evaluation on SPEC CPU To be able to compare all the programs in the benchmark a k-fold cross validation is performed: k disjoint subsets of benchmarks are created from SPEC CPU. Each subset becomes the test set of the model trained on the remaining part of the benchmarks. At each iteration the model is re-initialized and trained from scratch. For the purpose of this project k is chosen to be 10.

Model	Shared W	Norm.
A	✓	✓
B	✓	✗
C	✗	✓
D	✗	✗

Table 4.1: List of configurations of model's parameters. "Shared W" represents the sharing of the LSTM parameters and "Norm." is the normalization of the embeddings.

Model exploration Before evaluating the predictors performance, four different variations of model are compared, to then choose the model to be used later. The 4 models test all the permutations of 2 model parameters:

- **Sharing LSTM's parameter**: The 3 LSTMs in Figure 4.4 can be different network or be the same used in all 3 cases. Given that the task of the LSTM is to encode a basic block, and that the problem is somewhat symmetric it is plausible that a single LSTM is enough to do that.
- **The embedding normalization**: In the `inst2vec` paper [6] the embeddings are normalized before being used to perform predictions. The normalization transforms each embedding into a unitary length vector in the L_2 distance.

A model is built for each combination of these two feature obtaining the 4 models (identified by letters) summarized in Table 4.1. These models are trained on 15 out of the 20 benchmarks, and tested on the remaining 5. The split is performed randomly.

Chapter 5

Results

This Chapter shows the outcome of applying the model described in Chapter 4 on SPEC's CPU 2006. The evaluation follows a similar structure to Chapter 3 where prediction accuracy and its impact on the performance are evaluated separately.

5.1 Model Selection

The following section presents the results of model selection among the models of Table 4.1 and described in Subsection 4.3.2. Each model is trained and tested on the same sets of benchmarks: the test set is composed of the following five benchmarks: `473.astar` ,`456.hmmer` ,`458.sjeng` ,`483.xalancbmk`, and `445.gobmk`, while the remaining ones are further divided into training and validation sets. The validation set is chosen to have two benchmarks not to reduce the training set size excessively. The validation set is composed of the following benchmarks chosen at random: `400.perlbench`, and `482.sphinx4`.The training is done with early stopping [32] on the loss of the validation set.

The evaluation of each model is done on the metrics described in Section 3.3 and its results on the test set are summarized in Table 5.1. Each model has the best score on one of the four metrics, therefore it is not possible to conclude that a model performs better than the others. We restricted the choice between model A and B, as they are the network with less parameter. Among A and B we choose B as it has better score on three out of four metrics in Table 5.1.
We can get some insight on the trained model by looking at the distribution of the predictions. The left column of Figure 5.1 shows the distribution of the predicted branch probabilities on the test set compared to the ground truth

Model	Accuracy	MSE	WMSE	Miss Rate
A	60.91%	0.20015	**0.057**	16.86%
B	**63.12%**	0.19769	0.082	16.55%
C	60.64%	**0.19468**	0.079	17.61%
D	61.12%	0.20506	0.060	**15.41%**

Table 5.1: Summary of the metrics computed on the four models of Table 4.1 on the training set.

ones on the same set. The distribution of the prediction error is also shown on the right column. The red dashed line represents the average execution count of the branches in each error interval of the histogram.

From the distributions on the left columns one can qualitatively see that the predicted distribution assigns more mass to the extremes of the $[0, 1]$ interval than the LLVM's predictor. From the right column we observe the unimodal shape of the error distribution, and that the peak of the average execution count tends to be close to zero, indicating that frequently executed branches are easier to predict.

5.2 Prediction Result

To be able to compare all the benchmarks a k-fold cross validation is performed: k disjoint subsets of benchmarks are created from SPEC CPU then each subset becomes the test set of the model trained on the remaining part of the benchmarks. At each iteration the model is re-initialized and trained from scratch. For the purpose of this project k is chosen to be 10.

Table 5.3 presents the results for the accuracy and miss rate, while Table 5.3 displays the results for the MSE and WMSE. The tables compare the predictor with the results on the heuristics of Chapter 3. On a per-benchmark analysis, the developed branch predictor constantly outperforms the other approaches in accuracy, miss rate, and MSE for most of the programs. Overall the branch predictor improves Ball and Larus heuristic' accuracy of 4.69% and slightly improves the miss rate. Moreover the predictor has the lowers MSE, and the second best WMSE.

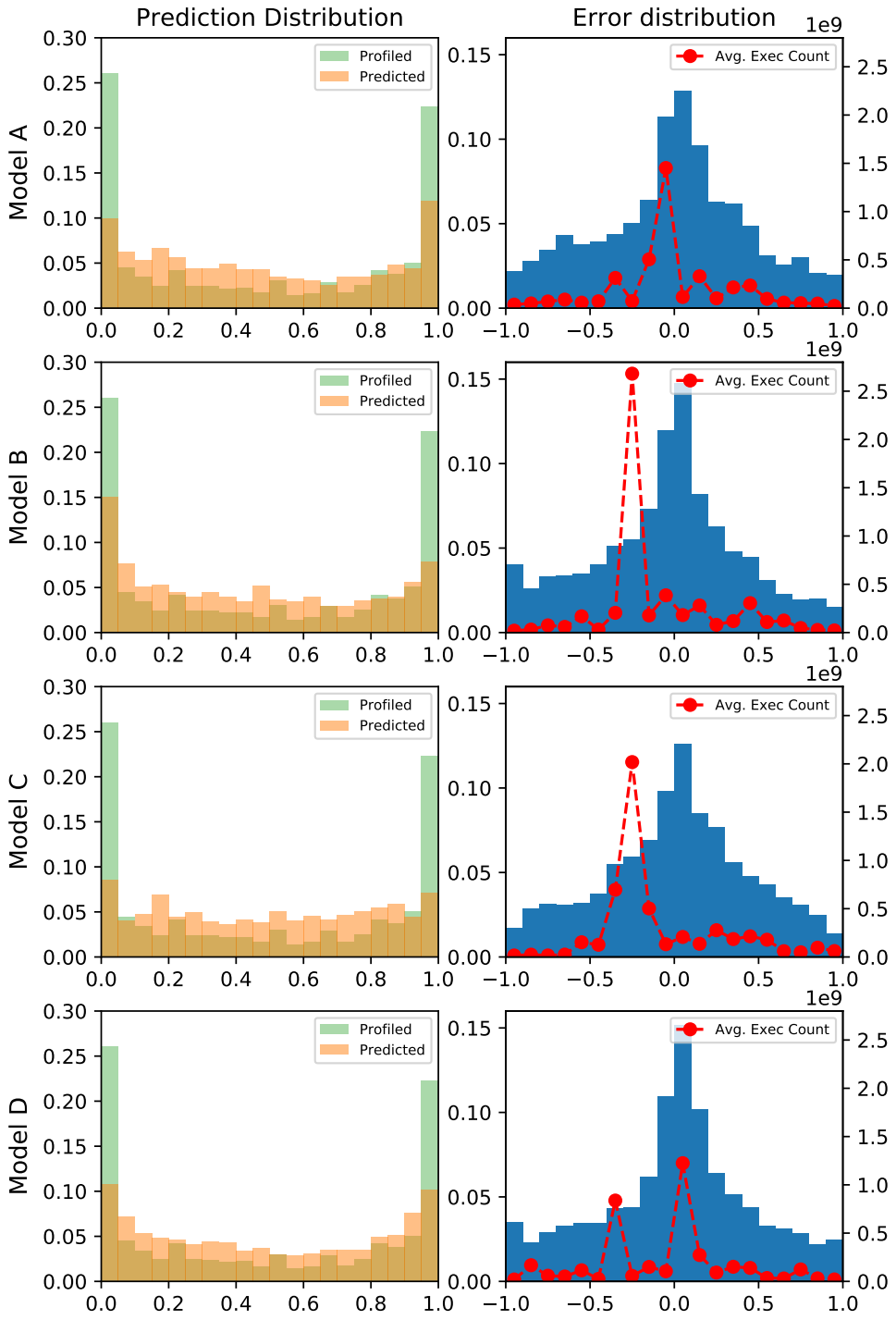


Figure 5.1: Prediction results of the four models of Table 4.1 on the test set.The left column show the distribution of the predicted and true branch probabilities on the test set, while the right one shows the distribution of the prediction error. In red is the weighted error distribution.

Benchmarks	Accuracy				Miss Rate				
	LLVM	BALL	WU	PRED	LLVM	BALL	WU	PRED	PBP
400.perlbench	49.845%	57.714%	56.284%	**60.990%**	28.093%	**17.906%**	20.544%	63.463%	5.199%
401.bzip2	67.077%	**76.572%**	75.215%	71.517%	23.665%	15.977%	28.833%	**15.106%**	9.717%
403.gcc	50.009%	58.203%	57.278%	**61.311%**	37.502%	28.580%	31.807%	**24.628%**	9.738%
429.mcf	**69.277%**	68.674%	66.867%	68.675%	**28.145%**	38.827%	29.590%	30.571%	13.750%
445.gobmk	53.005%	55.935%	55.213%	**60.452%**	41.832%	36.482%	39.890%	**35.806%**	16.830%
456.hmmer	61.704%	70.356%	63.613%	**76.972%**	62.794%	**19.659%**	19.787%	27.426%	9.161%
458.sjeng	51.203%	56.541%	58.646%	**60.150%**	38.079%	30.862%	**29.458%**	30.785%	15.531%
462.libquantum	69.406%	75.799%	75.342%	**80.822%**	34.673%	34.667%	34.679%	**30.448%**	11.528%
464.h264ref	55.006%	61.670%	56.887%	**71.365%**	33.521%	24.902%	28.907%	**23.195%**	14.363%
471.omnetpp	70.120%	69.609%	68.912%	**72.955%**	**39.542%**	41.541%	45.542%	42.028%	11.162%
473.astar	66.842%	72.631%	67.105%	**73.421%**	40.598%	**28.758%**	33.393%	37.345%	22.267%
483.xalancbmk	60.542%	58.312%	56.034%	**64.871%**	10.311%	15.056%	10.610%	**9.830%**	8.691%
433.milc	69.230%	71.400%	64.497%	**71.598%**	22.696%	22.651%	23.301%	**22.631%**	22.602%
444.namd	43.934%	49.398%	39.344%	**65.574%**	43.624%	**20.430%**	62.409%	41.083%	11.169%
447.dealII	70.730%	60.070%	56.769%	**76.709%**	**22.515%**	24.756%	26.372%	22.876%	21.811%
450.soplex	66.192%	**69.293%**	65.582%	67.819%	15.169%	10.385%	15.666%	**9.747%**	7.701%
453.povray	46.468%	56.270%	52.475%	**60.231%**	55.075%	49.330%	52.361%	**44.695%**	14.799%
470.lbm	63.157%	84.210%	77.192%	**84.211%**	61.641%	30.449%	30.502%	**4.290%**	4.289%
482.sphinx3	63.176%	**76.444%**	71.570%	75.090%	29.426%	21.746%	21.991%	**17.421%**	5.668%
Total	54.245%	59.137%	57.138%	**63.823%**	22.924 %	21.122%	20.798%	**20.459%**	12.508 %

Table 5.2: Accuracy and Miss Rate for the different branch probability predictors. The columns titled PRED are the predictor result.

Benchmarks	Mean Squared Error				Weighted Mean Squared Error			
	LLVM	BALL	WU	PRED	LLVM	BALL	WU	PRED
400.perlbench	0.187	0.187	0.197	**0.181**	0.111	**0.110**	0.171	0.272
401.bzip2	0.148	**0.126**	0.139	0.140	0.067	**0.067**	0.097	0.071
403.gcc	0.190	0.195	0.205	**0.184**	0.121	0.120	0.141	**0.102**
429.mcf	**0.125**	0.156	0.154	0.146	**0.114**	0.176	0.131	0.119
445.gobmk	0.151	0.180	0.191	**0.147**	0.128	0.121	0.134	**0.115**
456.hmmer	0.129	0.139	0.168	**0.102**	0.137	0.124	0.126	**0.106**
458.sjeng	0.154	0.182	0.185	**0.143**	**0.088**	0.101	0.098	0.099
462.libquantum	**0.082**	0.124	0.129	0.094	**0.068**	0.184	0.183	0.076
464.h264ref	0.154	0.181	0.193	**0.129**	**0.062**	0.078	0.089	0.063
471.omnetpp	**0.128**	0.160	0.160	0.128	0.155	0.195	0.190	**0.151**
473.astar	0.117	0.132	0.163	**0.107**	0.070	**0.061**	0.069	0.072
483.xalancbmk	0.161	0.193	0.210	**0.161**	0.025	0.065	**0.022**	0.059
433.milc	0.091	0.110	0.159	**0.080**	0.042	**0.006**	0.020	0.009
444.namd	0.227	0.260	0.281	**0.197**	**0.079**	0.083	0.180	0.135
447.dealII	0.150	0.193	0.205	**0.122**	0.060	0.031	0.030	**0.023**
450.soplex	**0.135**	0.160	0.164	0.146	**0.060**	0.066	0.078	0.065
453.povray	0.175	0.193	0.231	**0.150**	0.216	0.200	0.243	**0.139**
470.lbm	0.114	0.104	0.119	**0.087**	0.126	0.124	0.125	**0.091**
482.sphinx3	0.123	0.121	0.132	**0.107**	**0.079**	0.135	0.153	0.087
Total	0.173	0.187	0.199	**0.164**	**0.059**	0.076	0.064	0.071

Table 5.3: Mean Squared Error and Weighted Mean Squared Error for the different branch probability predictors. The columns titled PRED are the predictor result.

5.3 Performance impact

The effectiveness of the branch predictor is finally measured as the impact it has on the performance of the programs. Each benchmark is compiled with the predicted branch probability obtained in the previous sections. The compilation and running parameters are the same used in Chapter 3. The running time of each benchmark is compared against LLVM's branch predictor to obtain the speedup as described by equation 3.8.

The results are shown in Figure 5.3 (without LTO) and 5.4 (with LTO). On average the predictor has a slight slowdown compared to LLVM's predictor both with and without LTO. Turning the attention to the per-benchmark speedup we observe the predictor is constantly outperformed by Ball and Larus and Wu and Larus heuristics both with and without LTO.

Similar to the analysis of Figure 5.1, Figure 5.2 shows the distribution of the predicted branch probabilities and the prediction error. Qualitative insights can be drawn from the figure: the distribution of branch probability is very uniform apart from two peaks near 0 and 1. Moreover the error distribution is consistent with the ones obtained in Figure 5.1. Also note that the peak of the average execution count is exactly in 0.

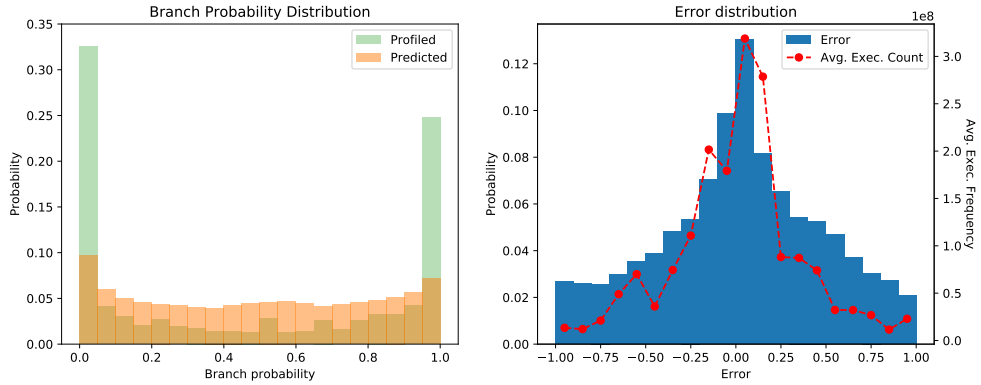


Figure 5.2: on the left, distribution of the predicted branch probabilities for all the programs in SPEC CPU. On the right the distribution of the prediction error and it relationship with execution counts.

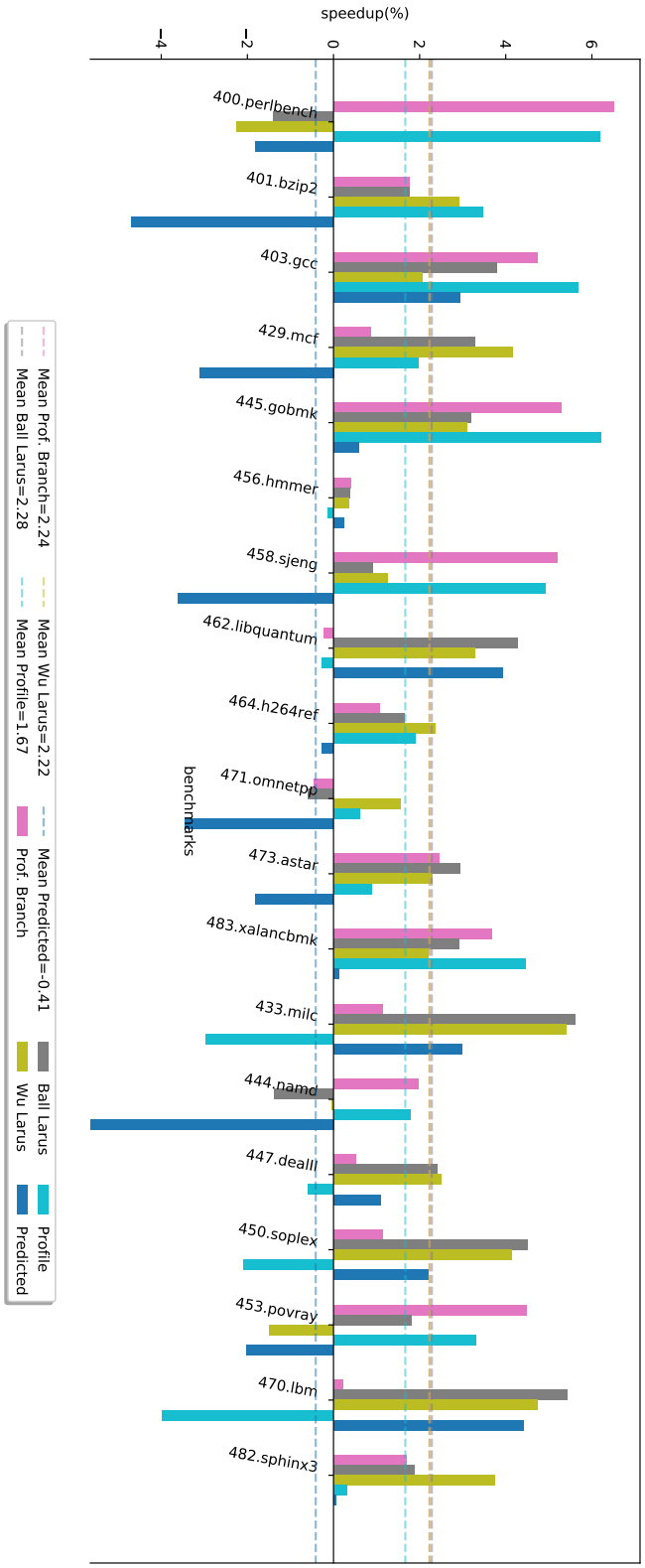


Figure 5.3: Speedups obtained for the different branch predictors compared to the base LLVM predictor.

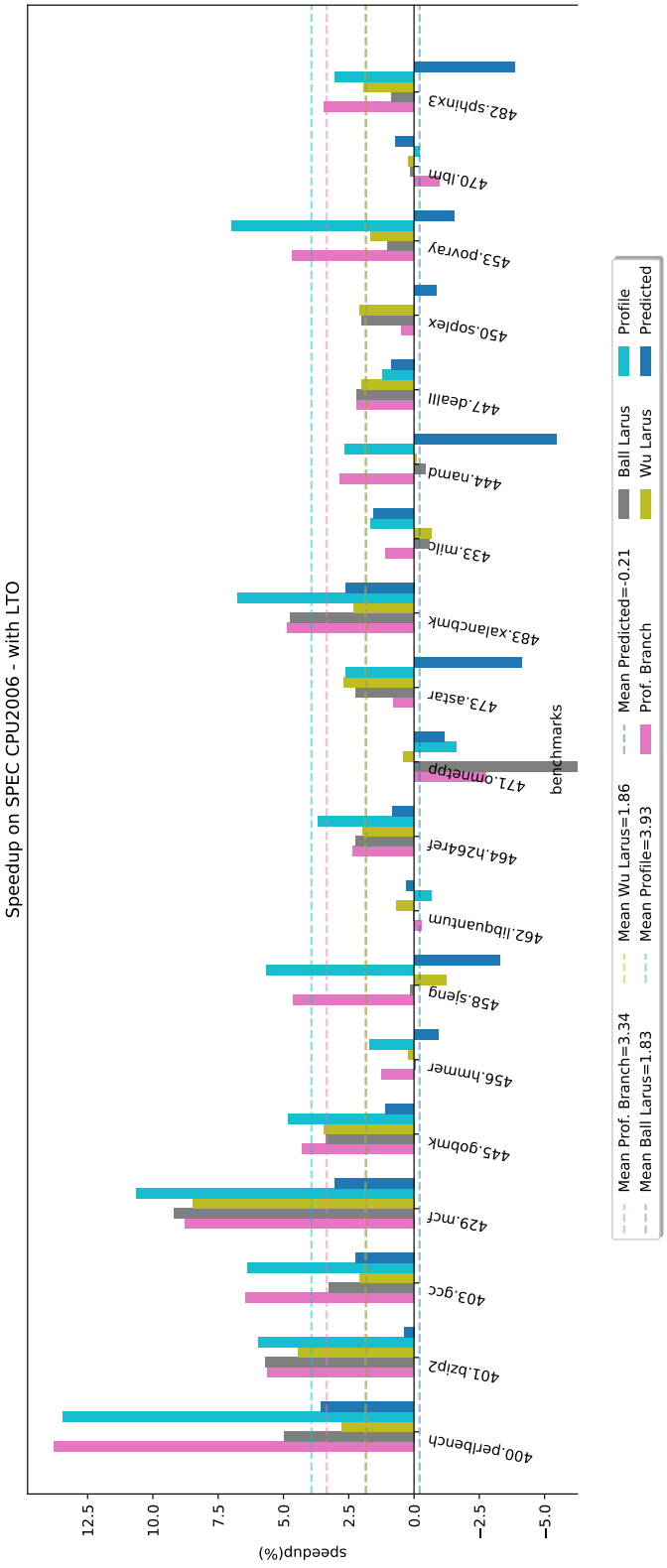


Figure 5.4: Speedups obtained for the different branch predictors with Link Time Optimization enabled compared to the base LLVM predictor.

Chapter 6

Conclusion and Discussion

This chapter draws the conclusion from the results presented in Chapter 5. The research question presented in Chapter 1 is answered and finally future work is proposed.

6.1 Conclusion

In Chapter 4 we developed a branch predictor that uses representation learning to encode the instructions of a LLVM IR program. The predictor is evaluated as a branch predictor and as a branch probability predictor against the state-of-the-art heuristics. Table 5.2 shows that the developed predictor outperforms the state-of-the-art heuristics in branch prediction obtaining a smaller miss rate and a higher accuracy. As a branch probability predictor obtains the smallest MSE among the considered predictors, but not the WMSE. Section 5.3 analyzes the speedup obtained with the predictor compared to LLVM's one. It shows that, despite having the lowest MSE, accuracy, and miss rate, the predictor causes on average slowdowns compared to LLVM's and it is consistently outperformed by Ball and Larus and Wu and Larus heuristics. The use of link time optimization (LTO) leads to similar results and conclusions.

Addressing the research question, representation learning allows for a more accurate branch probability prediction compared the considered heuristics as displayed by the improvements of MSE, accuracy, and miss rate. However, the predictor does not obtain the same improvements on the speedup of the compiled programs. For this reason, and for the computational cost of using a neural network, the heuristics are still more suitable for LLVM. However the predictor showed positive results for some programs in the benchmark, therefore we advocate further research to draw decisive conclusions. The next

section suggests some ideas for future work.

Threats to validity The project takes into account only SPEC's CPU benchmark which is purposely crafted to stress CPU and memory. The benchmark may not represent the behaviour of most programs, which can undermine the conclusion drawn in this section. Cummins *et al.* [24] pose the problem of the small amount of datasets used in compiler research and its negative effect on the machine learning models.

6.2 Future Work

The conclusions drawn in this thesis are not decisive, therefore to consolidate the work more experimentation must be performed. The following paragraphs suggest research ideas and experiment to continue this work.

The first consideration derives from the threats to validity described in the previous section, diverse sets of programs and benchmarks must be used to draw general conclusion. Optimizing the branch predictor is another natural direction future research can approach. The optimization can target two aspects of the branch predictor: improving the representation of the input, and fine tuning the classifier. The former seeks for a better representation method for the instruction, while the latter optimizes the classifier in prediction performance. Specifically, `inst2vec` can be modified to generalize better across different programs or different representation of code can be devised. For example it is possible embed the single tokens in each instruction instead of the whole instruction. Current research is also experimenting with representation of programs as graphs [3, 2]. A recent one uses graph neural networks [25] to perform program analysis and optimization. Possible improvements can also be achieved by modifying the loss function to take into account the execution count of the branches. It is also possible to build and evaluate hybrid branch predictors combining heuristics and a machine learning powered predictor similar to the work of Veerle *et al.* [26]. The two can be combined by using Dempster-Shafer theory, or by prioritizing heuristics when applicable and then using machine learning to predict the other branches.

Current compilers are focused on a representation of the code that is functional to the different stages of compilation, but little focus is given to a representation can be used for predictive modeling. The first attempt to add a second representation of the code was made with `MilepostGCC` [30]. `MilepostGCC` is a compiler based on GCC that extracts static feature from source code,

and uses machine learning models to predict the optimal optimization flags for compilation. The features are human-engineered and are easily extracted from the source code and the CFG. `MilepostGCC` proved a relevant and consistent speedup over GCC [30]. The use of other representations of code such as `inst2vec` has proven useful within single tasks such as heterogenous device mapping [6] and thread coarsening [52, 23], but no compiler has been developed to incorporate it into its workflow. Therefore a possible future work is in developing such a compiler, schematized in Figure 6.1, where a parallel representation of the compiler's IR is used to perform analysis on the code and as a feedback to the optimization process. A similar vision is expressed in the work of Cummins *et al*. [23] were they show how their optimization predictor for OpenCL kernels could be part of such architecture. Although they propose to learn features directly from the source code, which is impractical as it implies developing and training a model for each source language. Using neural networks in the prediction part in the architecture of Figure 6.1 might look computational demanding, but note that the network can be used to output multiple prediction at the same time, therefore amortizing the cost of prediction. The predictor developed in this thesis is one of the many architecture fitting for such a compiler structure.

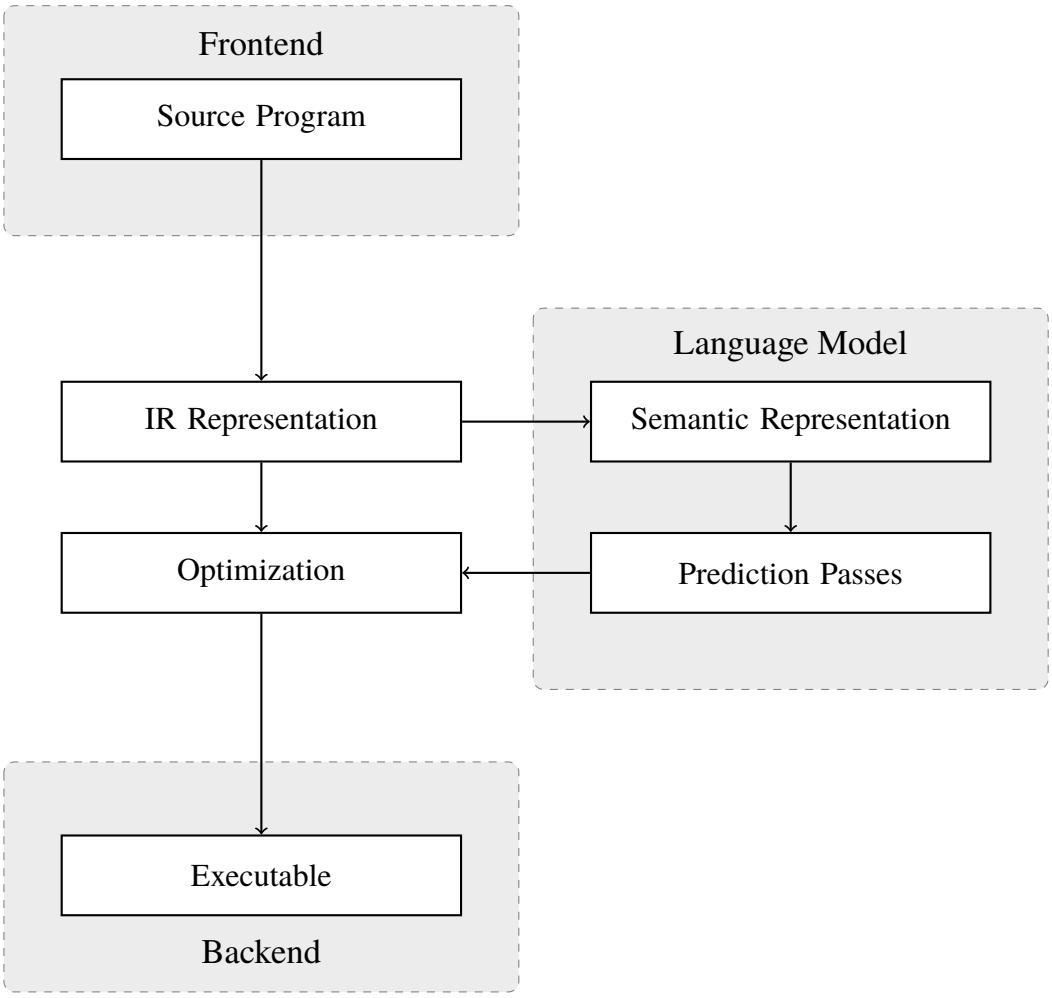


Figure 6.1: High level structure of a compiler that uses a parallel representation of the program to do automatic analysis and optimization of the code.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. *Learning to Represent Programs with Graphs*. 2017. arXiv: 1711.00740 [cs.LG].

[3] Miltiadis Allamanis et al. *A Survey of Machine Learning for Big Code and Naturalness*. 2017. arXiv: 1709.06182 [cs.SE].

[4] Thomas Ball and James R. Larus. "Branch Prediction for Free". In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: ACM, 1993, pp. 300–313. ISBN: 0-89791-598-4. DOI: 10.1145/155090.155119.

[5] Anton Beloglazov et al. "Chapter 3 - A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems". In: ed. by Marvin V. Zelkowitz. Vol. 82. Advances in Computers. Elsevier, 2011, pp. 47–111. DOI: https://doi.org/10.1016/B978-0-12-385512-1.00003-7.

[6] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 3588–3600. URL: http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf.

[7] Y. Bengio, A. Courville, and P. Vincent. "Representation Learning: A Review and New Perspectives". In: *IEEE Transactions on Pattern*

Analysis and Machine Intelligence 35.8 (Aug. 2013), pp. 1798–1828. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2013.50.

[8] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[9] L. Benini et al. "Regression Models for Behavioral Power Estimation". In: *Integr. Comput.-Aided Eng.* 5.2 (Apr. 1998), pp. 95–106. ISSN: 1069-2509. URL: http://dl.acm.org/citation.cfm?id=1275815.1275817.

[10] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[11] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646.

[12] R. P. L. Buse and W. Weimer. "The road not taken: Estimating path execution frequency statically". In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 144–154. DOI: 10.1109/ICSE.2009.5070516.

[13] Brad Calder et al. "Corpus-based Static Branch Prediction". In: *SIGPLAN Not.* 30.6 (June 1995), pp. 79–92. ISSN: 0362-1340. DOI: 10.1145/223428.207118.

[14] Brad Calder et al. "Evidence-based Static Branch Prediction Using Machine Learning". In: *ACM Trans. Program. Lang. Syst.* 19.1 (Jan. 1997), pp. 188–222. ISSN: 0164-0925. DOI: 10.1145/239912.239923.

[15] E. Cambria and B. White. "Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]". In: *IEEE Computational Intelligence Magazine* 9.2 (2014), pp. 48–57.

[16] Erik Cambria and Bebo White. "Jumping NLP curves: A review of natural language processing research". In: *IEEE Computational intelligence magazine* 9.2 (2014), pp. 48–57.

[17] Cagatay Catal and Banu Diri. “A systematic review of software fault prediction studies”. In: *Expert Systems with Applications* 36.4 (2009), pp. 7346–7354. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2008.10.027.

[18] J. Cavazos et al. “Rapidly Selecting Good Compiler Optimizations using Performance Counters”. In: *International Symposium on Code Generation and Optimization (CGO’07)*. Mar. 2007, pp. 185–197. DOI: 10.1109/CGO.2007.32.

[19] E. Ceylan, F. O. Kutlubay, and A. B. Bener. “Software Defect Identification Using Machine Learning Techniques”. In: *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO’06)*. 2006, pp. 240–247.

[20] Gobinda G. Chowdhury. “Natural language processing”. In: *Annual Review of Information Science and Technology* 37.1 (2003), pp. 51–89. DOI: 10.1002/aris.1440370103.

[21] Yu-An Chung and James Glass. “Speech2Vec: A Sequence-to-Sequence Framework for Learning Word Embeddings from Speech”. In: *Interspeech 2018* (Sept. 2018). DOI: 10.21437/interspeech.2018-2341.

[22] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[23] C. Cummins et al. “End-to-End Deep Learning of Optimization Heuristics”. In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2017, pp. 219–232.

[24] C. Cummins et al. “Synthesizing benchmarks for predictive modeling”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, pp. 86–99.

[25] Chris Cummins et al. *ProGraML: Graph-based Deep Learning for Program Optimization and Analysis*. 2020. arXiv: 2003.10536 [cs.LG].

[26] Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. “Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction”. In: *Advances in Computer Systems Architecture*. Ed. by Thambipillai Srikanthan, Jingling Xue, and Chip-Hong Chang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 336–352. ISBN: 978-3-540-32108-8.

[27] Jingcheng Du et al. “Gene2Vec: Distributed Representation of Genes Based on Co-Expression”. In: *bioRxiv* (2018). DOI: 10.1101/286096.

[28] Andre Esteva et al. “A guide to deep learning in healthcare”. In: *Nature Medicine* 25.1 (2019), pp. 24–29. ISSN: 1546-170X. DOI: 10.1038/s41591-018-0316-z.

[29] Joseph A. Fisher and Stefan M. Freudenberger. “Predicting Conditional Branch Directions from Previous Runs of a Program”. In: *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS V. Boston, Massachusetts, USA: ACM, 1992, pp. 85–95. ISBN: 0-89791-534-8. DOI: 10.1145/143365.143493.

[30] Grigori Fursin et al. “Milepost GCC: Machine Learning Enabled Self-tuning Compiler”. In: *International Journal of Parallel Programming* 39 (3 2011), pp. 296–327. ISSN: 0885-7458. DOI: 10.1007/s10766-010-0161-2.

[31] Yoav Goldberg and Omer Levy. *word2vec Explained: deriving Mikolov et al.’s negative-sampling word-embedding method*. 2014. arXiv: 1402.3722 [cs.CL].

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 0262035618, 9780262035613.

[33] Klaus Greff et al. “LSTM: A Search Space Odyssey”. In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (Oct. 2017), pp. 2222–2232. ISSN: 2162-2388. DOI: 10.1109/tnnls.2016.2582924.

[34] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. AMS, 2003. ISBN: 978-0821807491, 0821807498.

[35] Aditya Grover and Jure Leskovec. “Node2vec: Scalable Feature Learning for Networks”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 855–864. ISBN: 9781450342322. DOI: 10.1145/2939672.2939754.

[36] Dan Guest, Kyle Cranmer, and Daniel Whiteson. “Deep Learning and Its Application to LHC Physics”. In: *Annual Review of Nuclear and Particle Science* 68.1 (2018), pp. 161–181. DOI: 10.1146/annurev-nucl-101917-021019.

[37] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055, 9780128119051.

[38] John Henning. *SPEC CPU2006 Documentation*. English. URL: www.spec.org/cpu2006/Docs/.

[39] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737.

[40] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[41] Kurt Hornik. "Approximation Capabilities of Multilayer Feedforward Networks". In: *Neural Netw.* 4.2 (Mar. 1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T.

[42] Srinivasan Iyer et al. "Summarizing Source Code using a Neural Attention Model". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. DOI: 10.18653/v1/P16-1195.

[43] D. A. Jimenez and C. Lin. "Perceptron learning for predicting the behavior of conditional branches". In: *IJCNN'01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222)*. Vol. 3. July 2001, 2122–2127 vol.3. DOI: 10.1109/IJCNN.2001.938494.

[44] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[45] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Washington, DC, USA: IEEE Computer Society, Mar. 2004, pp. 75–86. ISBN: 0-7695-2102-9. DOI: 10.1109/CGO.2004.1281665.

[46] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[47] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems". In: *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 30. Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 330–335. ISBN: 0818679778.

[48] Xianhua Liu et al. "Basic-block Reordering Using Neural Networks". In: (Jan. 2007).

[49] Roberto Castaneda Lozano. *Unison Manual*. English. `https://unison-code.github.io/doc/manual.pdf`. 2019.

[50] Roberto Castañeda Lozano et al. "Constraint-Based Register Allocation and Instruction Scheduling". In: *Principles and Practice of Constraint Programming*. Ed. by Michela Milano. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 750–766. ISBN: 978-3-642-33558-7.

[51] A. Magni, C. Dubach, and M. O'Boyle. "Automatic optimization of thread-coarsening for graphics processors". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Aug. 2014, pp. 455–466. DOI: 10.1145/2628071.2628087.

[52] Alberto Magni, Christophe Dubach, and Michael O'Boyle. "Automatic Optimization of Thread-Coarsening for Graphics Processors". In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14. Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 455–466. ISBN: 9781450328098. DOI: 10.1145/2628071.2628087.

[53] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[54] S. McFarling and J. Hennesey. "Reducing the Cost of Branches". In: *SIGARCH Comput. Archit. News* 14.2 (May 1986), pp. 396–403. ISSN: 0163-5964. DOI: 10.1145/17356.17402.

[55] Ivan Medennikov and Anna Bulusheva. "LSTM-Based Language Models for Spontaneous Speech Recognition". In: *Speech and Computer*. Ed. by Andrey Ronzhin, Rodmonga Potapova, and Géza Németh. Cham: Springer International Publishing, 2016, pp. 469–475. ISBN: 978-3-319-43958-7.

[56] Charith Mendis et al. “Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, Sept. 2019, pp. 4505–4515. URL: http://proceedings.mlr.press/v97/mendis19a.html.

[57] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: 1310.4546 [cs.CL].

[58] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[59] Sparsh Mittal. “A survey of techniques for dynamic branch prediction”. In: *Concurrency and Computation: Practice and Experience* 31.1 (2019), e4666. DOI: 10.1002/cpe.4666.

[60] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020, 9780262018029.

[61] Yusuke Oda et al. “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T)”. In: Nov. 2015, pp. 574–584. DOI: 10.1109/ASE.2015.36.

[62] Christine Payne. *MuseNet*. 2019. URL: http://openai.com/blog/musenet.

[63] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “Glove: Global vectors for word representation”. In: *In EMNLP*. 2014.

[64] Erez Perelman et al. “Using SimPoint for Accurate and Efficient Simulation”. In: *SIGMETRICS Perform. Eval. Rev.* 31.1 (June 2003), pp. 318–319. ISSN: 0163-5999. DOI: 10.1145/885651.781076. URL: https://doi.org/10.1145/885651.781076.

[65] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. “Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite”. In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), pp. 412–423. ISSN: 0163-5964. DOI: 10.1145/1273440.1250713.

[66] Edgar Allan Poe and ST Joshi. *The Raven: Tales and Poems*. Penguin Classics, 2013.

[67] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. 2019.

[68] G. Ramalingam. “Data Flow Frequency Analysis”. In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 267–277. ISBN: 0-89791-795-2. DOI: 10.1145/231379.231433.

[69] C. V. Ramamoorthy. “Analysis of Computational Systems: Discrete Markov Analysis of Computer Programs”. In: *Proceedings of the 1965 20th National Conference*. ACM ’65. Cleveland, Ohio, USA: ACM, 1965, pp. 386–392. DOI: 10.1145/800197.806061.

[70] Stefano Crespi Reghizzi, Luca Breveglieri, and Angelo Morzenti. *Formal Languages and Compilation*. 2nd. Springer Publishing Company, Incorporated, 2013. ISBN: 1447155130, 9781447155133. DOI: 10.1007/978-1-84882-050-0.

[71] Xin Rong. *word2vec Parameter Learning Explained*. 2014. arXiv: 1411.2738 [cs.CL].

[72] Suyog Sarda and Mayur Pandey. *LLVM Essentials*. Packt Publishing, 2015. ISBN: 1785280805.

[73] V. Sarkar. “Determining Average Program Execution Times and Their Variance”. In: *SIGPLAN Not.* 24.7 (June 1989), pp. 298–312. ISSN: 0362-1340. DOI: 10.1145/74818.74845.

[74] F. Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (Jan. 2009), pp. 61–80. ISSN: 1941-0093. DOI: 10.1109/TNN.2008.2005605.

[75] R. Sethuram et al. “A Neural Net Branch Predictor to Reduce Power”. In: *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID’07)*. Jan. 2007, pp. 679–684. DOI: 10.1109/VLSID.2007.14.

[76] A. Shabtai, Y. Fledel, and Y. Elovici. “Automated Static Code Analysis for Classifying Android Applications Using Machine Learning”. In: *2010 International Conference on Computational Intelligence and Security*. Dec. 2010, pp. 329–333. DOI: 10.1109/CIS.2010.77.

[77] R. Sheikh, J. Tuck, and E. Rotenberg. “Control-Flow Decoupling: An Approach for Timely, Non-Speculative Branching”. In: *IEEE Transactions on Computers* 64.8 (2015), pp. 2182–2203.

[78] Alex Sherstinsky. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network”. In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306. URL: http://dx.doi.org/10.1016/j.physd.2019.132306.

[79] T. Sherwood, E. Perelman, and B. Calder. “Basic block distribution analysis to find periodic behavior and simulation points in applications”. In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. Sept. 2001, pp. 3–14. DOI: 10.1109/PACT.2001.953283.

[80] Timothy Sherwood et al. “Automatically Characterizing Large Scale Program Behavior”. In: *SIGOPS Oper. Syst. Rev.* 36.5 (Oct. 2002), pp. 45–57. ISSN: 0163-5980. DOI: 10.1145/635508.605403.

[81] Zhan Shi et al. *Learning Execution through Neural Code Fusion*. 2019. arXiv: 1906.07181 [cs.LG].

[82] Daniel Soutner and Luděk Müller. “Application of LSTM Neural Networks in Language Modelling”. In: *Text, Speech, and Dialogue*. Ed. by Ivan Habernal and Václav Matoušek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 105–112. ISBN: 978-3-642-40585-3.

[83] Cloyce D. Spradling. “SPEC CPU2006 Benchmark Tools”. In: *SIGARCH Comput. Archit. News* 35.1 (Mar. 2007), pp. 130–134. ISSN: 0163-5964. DOI: 10.1145/1241601.1241625.

[84] Y. N. Srikant and Priti Shankar. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2007. ISBN: 142004382X, 9781420043822.

[85] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 144141276X.

[86] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009. ISBN: 144141276X, 9781441412768.

[87] M. Sundermeyer, H. Ney, and R. Schlüter. “From Feedforward to Recurrent LSTM Neural Networks for Language Modeling”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23.3 (2015), pp. 517–529.

[88] Clang Team. *Clang: A C language family frontend for LLVM*. clang.llvm.org. 2019.

[89] D. Tetzlaff and S. Glesner. "Static Prediction of Loop Iteration Counts Using Machine Learning to Enable Hot Spot Optimizations". In: *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. Sept. 2013, pp. 300–307. DOI: 10.1109/SEAA.2013.12.

[90] Tim A. Wagner et al. "Accurate Static Estimators for Program Optimization". In: *SIGPLAN Not.* 29.6 (June 1994), pp. 85–96. ISSN: 0362-1340. DOI: 10.1145/773473.178251.

[91] David W. Wall. "Predicting Program Behavior Using Real or Estimated Profiles". In: *SIGPLAN Not.* 26.6 (May 1991), pp. 59–70. ISSN: 0362-1340. DOI: 10.1145/113446.113451.

[92] Shuohang Wang and Jing Jiang. "Learning Natural Language Inference with LSTM". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2016). DOI: 10.18653/v1/n16-1170.

[93] Z. Wang and M. O'Boyle. "Machine Learning in Compiler Optimization". In: *Proceedings of the IEEE* 106.11 (Nov. 2018), pp. 1879–1901. ISSN: 1558-2256. DOI: 10.1109/JPROC.2018.2817118.

[94] Niklaus Wirth. *Algorithms + Data Structures = Programs*. USA: Prentice Hall PTR, 1978. ISBN: 0130224189.

[95] W. F. Wong. "Source Level Static Branch Prediction". In: *The Computer Journal* 42.2 (Jan. 1999), pp. 142–149. ISSN: 0010-4620. DOI: 10.1093/comjnl/42.2.142.

[96] Youfeng Wu and James R. Larus. "Static Branch Frequency and Program Profile Analysis". In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: ACM, 1994, pp. 1–11. ISBN: 0-89791-707-3. DOI: 10.1145/192724.192725.

[97] Zonghan Wu et al. *A Comprehensive Survey on Graph Neural Networks*. 2019. arXiv: 1901.00596 [cs.LG].

[98] Ronald R. Yager and Naif Alajlan. “Dempster-Shafer Belief Structures for Decision Making Under Uncertainty”. In: *Know.-Based Syst.* 80.C (May 2015), pp. 58–66. ISSN: 0950-7051. DOI: 10.1016/j.knosys.2014.12.031.

[99] Pengcheng Yin and Graham Neubig. “A Syntactic Neural Model for General-Purpose Code Generation”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2017). DOI: 10.18653/v1/p17-1041.

[100] T. Young et al. “Recent Trends in Deep Learning Based Natural Language Processing [Review Article]”. In: *IEEE Computational Intelligence Magazine* 13.3 (2018), pp. 55–75.

[101] Yuwei Zhang et al. “A variable-level automated defect identification model based on machine learning”. In: *Soft Computing* 24.2 (Jan. 2020), pp. 1045–1061. ISSN: 1433-7479. DOI: 10.1007/s00500-019-03942-3.

[102] Victor Zhong, Caiming Xiong, and Richard Socher. *Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning*. 2017. arXiv: 1709.00103 [cs.CL].

Appendix A

LSTM

This appendix focuses on the Long Short Term Memory (LSTM) [40] cells, a type of RNN (see Section 2.4) used in this thesis in Chapter 4.

The network learns to produce other vectors, called *gates*, they choose which features to remember, to forget and to output. The gates are: the forget gate (f_t) chooses how to filter the previous context vector, the input gate (i_t) chooses which feature to add to the context vector, and the output gate (o_t) filters which part of the context are output in h_t. The 'filtering' is performed with the Hadamard product ($\odot$) which is defined as the element-wise product of two multidimensional arrays. When used in a Hadamard product, i_t, f_t, o_t act like a switch on the elements of the context, since they all are in the $[0, 1]$ range. The LSTM cell is depicted in Figure A.1 and its equations in A.1.

$$
\begin{aligned}
\mathbf{f_t} &= \sigma(\hat{\mathbf{f}}_\mathbf{t}) = \sigma(\mathbf{W_f} \cdot \mathbf{x_t} + \mathbf{U_f} \cdot \mathbf{h_{t-1}} + \mathbf{b_f}) \\
\mathbf{i_t} &= \sigma(\hat{\mathbf{i}}_\mathbf{t}) = \sigma(\mathbf{W_i} \cdot \mathbf{x_t} + \mathbf{U_i} \cdot \mathbf{h_{t-1}} + \mathbf{b_i}) \\
\mathbf{o_t} &= \sigma(\hat{\mathbf{o}}_\mathbf{t}) = \sigma(\mathbf{W_o} \cdot \mathbf{x_t} + \mathbf{U_o} \cdot \mathbf{h_{t-1}} + \mathbf{b_o}) \\
\mathbf{a_t} &= \sigma(\hat{\mathbf{a}}_\mathbf{t}) = \tanh(\mathbf{W_c} \cdot \mathbf{x_t} + \mathbf{U_c} \cdot \mathbf{h_{t-1}} + \mathbf{b_c}) \\
\mathbf{c_t} &= \mathbf{f_t} \odot \mathbf{c_{t-1}} + \mathbf{i_t} \odot \mathbf{a_t} \\
\mathbf{h_t} &= \mathbf{o_t} \odot \tanh(\mathbf{c_t})
\end{aligned}
\tag{A.1}
$$

Where the function σ represents the sigmoid function and $\tanh$ is the hyperbolic tangent function.

Learning a model means finding values for the free parameters of the equations above, in particular for $\mathbf{W_f}$, $\mathbf{W_i}$, $\mathbf{W_o}$, $\mathbf{W_c}$, $\mathbf{U_f}$, $\mathbf{U_i}$, $\mathbf{U_o}$, $\mathbf{U_c}$, $\mathbf{b_f}$, $\mathbf{b_i}$, $\mathbf{b_o}$, $\mathbf{b_c}$. The number of parameters to learn in the LSTM's setting is much greater than the simple RNN case exposed in Subsection 2.4.2. Insights on how to fine-tune the network can be found in the following paper [33].

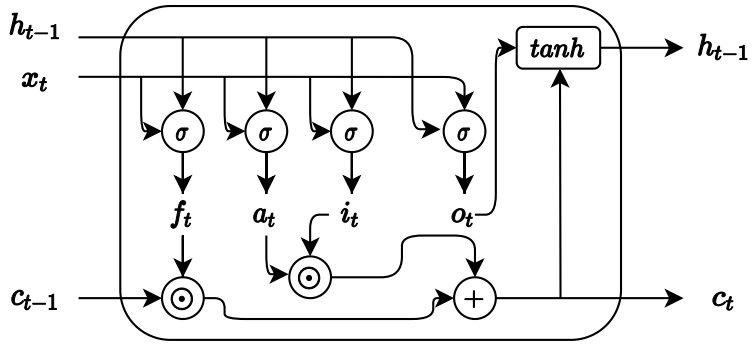


Figure A.1: Internals of a LSTM cell. The picture highlights the relationship between the different parts of the cell.

Appendix B

Complementary Results

benchmarks	LLVM	Prof. Branch	Ball Larus	Wu Larus	Profile	Predicted
400.perlbench	241.26	226.51	244.62	246.71	227.19	245.64
401.bzip2	372.40	365.94	365.98	361.89	359.92	389.85
403.gcc	236.39	225.68	227.77	231.63	223.68	229.64
429.mcf	272.26	269.94	263.59	261.38	266.99	280.74
445.gobmk	387.11	367.69	375.13	375.49	364.49	384.80
456.hmmer	302.16	300.92	301.00	301.06	302.55	301.45
458.sjeng	392.39	372.99	388.89	387.52	374.03	406.58
462.libquantum	446.42	447.44	428.13	432.22	447.64	429.57
464.h264ref	407.30	402.96	400.68	397.94	399.70	408.36
471.omnetpp	313.47	314.87	315.31	308.69	311.56	324.28
473.astar	322.51	314.77	313.27	315.28	319.63	328.36
483.xalancbmk	179.32	172.96	174.25	175.47	171.68	179.08
433.milc	400.76	396.26	379.44	380.18	412.67	389.16
444.namd	277.54	272.17	281.35	277.63	272.66	293.21
447.dealII	230.75	229.54	225.34	225.13	232.10	228.23
450.soplex	222.79	220.30	213.20	213.95	227.44	218.00
453.povray	112.67	107.83	110.66	114.34	109.07	114.95
470.lbm	314.27	313.61	298.08	300.04	326.79	300.99
482.sphinx3	407.32	400.50	399.79	392.60	406.01	407.10

Table B.1: Execution time for each benchmark, for the different testing configurations without LTO enabled.

benchmarks	LLVM	Prof. Branch	Ball Larus	Wu Larus	Profile	Predicted
400.perlbench	248.44	218.32	236.64	241.77	219.01	239.91
401.bzip2	387.17	366.55	366.30	370.75	365.36	385.67
403.gcc	231.91	217.89	224.60	227.18	217.99	226.81
429.mcf	280.03	257.42	256.48	258.15	253.12	271.75
445.gobmk	370.61	355.38	358.48	358.20	353.57	366.61
456.hmmer	304.73	301.02	304.86	304.06	299.65	307.62
458.sjeng	376.02	359.44	375.53	380.70	355.92	388.39
462.libquantum	429.66	430.86	429.55	426.75	432.48	428.41
464.h264ref	402.36	393.17	393.60	394.60	388.12	399.11
471.omnetpp	284.48	292.34	302.28	283.33	289.12	287.81
473.astar	311.41	309.00	304.63	303.28	303.47	324.22
483.xalancbmk	169.66	161.81	162.00	165.83	158.92	165.32
433.milc	365.53	361.56	367.66	367.93	359.61	360.01
444.namd	279.04	271.30	280.24	279.35	271.81	294.25
447.dealII	197.54	193.27	193.27	193.68	195.22	195.82
450.soplex	218.07	216.98	213.76	213.59	218.09	219.95
453.povray	101.87	97.33	100.84	100.21	95.23	103.44
470.lbm	300.05	302.92	299.60	299.40	300.64	297.91
482.sphinx3	401.10	387.78	397.59	393.56	389.27	416.59

Table B.2: Execution time for each benchmark, for the different testing configurations with LTO enabled.