VESPA: Static Profiling for Binary Optimization

ANGÉLICA APARECIDA MOREIRA, UFMG, Brazil
GUILHERME OTTONI, Facebook, Inc., USA
FERNANDO MAGNO QUINTÃO PEREIRA, UFMG, Brazil

Over the past few years, there has been a surge in the popularity of binary optimizers such as BOLT, Propeller, Janus and HALO. These tools use dynamic profiling information to make optimization decisions. Although effective, gathering runtime data presents developers with inconveniences such as unrepresentative inputs, the need to accommodate software modifications, and longer build times. In this paper, we revisit the static profiling technique proposed by Calder *et al.* in the late 90's, and investigate its application to drive binary optimizations, in the context of the BOLT binary optimizer, as a replacement for dynamic profiling. A few core modifications to Calder *et al.'s* original proposal, consisting of new program features and a new regression model, are sufficient to enable some of the gains obtained through runtime profiling. An evaluation of BOLT powered by our static profiler on four large benchmarks (clang, GCC, MySQL and PostgreSQL) yields binaries that are 5.47% faster than the executables produced by clang -O3.

CCS Concepts: • **Software and its engineering → Compilers**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Compiler, Optimization, Profiling, Prediction

ACM Reference Format:
Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: Static Profiling for Binary Optimization. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 144 (October 2021), 28 pages. https://doi.org/10.1145/3485521

1 INTRODUCTION

Feedback-driven optimization (FDO) has proven to be a valuable approach to improve the performance of programs beyond what static compilers can typically achieve [Chen et al. 2016; Hölzle and Ungar 1994; Li et al. 2010; Ottoni 2018; Panchenko et al. 2019, 2021]. In this scenario, the compiler uses information acquired from previous executions of the target program to perform more aggressive code transformations. FDO has been a key component of binary optimization tools, which rely on profiling information to carry out transformations like basic-block and function reordering. These tools achieve impressive results on top of highly optimized code: BOLT [Panchenko et al. 2019] claims speedups of up to 30%, and Propeller [Tallam 2019] reports 9%.

These results, however, come at a cost. Binary optimization relies on high-quality dynamic profiling information, collected during previous executions of the program. Gathering such data might represent a cumbersome step in the development cycle. It may require recompilation to add instrumentation in the target code. It might also require intermediate deployments in the production environment to collect data. Recompilation and early deployment can be prohibitively

Authors' addresses: Angélica Aparecida Moreira, Computer Science, UFMG, Belo Horizonte, Minas Gerais, Brazil, angelica.moreira@dcc.ufmg.br; Guilherme Ottoni, Facebook, Inc., Menlo Park, California, USA, ottoni@fb.com; Fernando Magno Quintão Pereira, Computer Science, UFMG, Belo Horizonte, Minas Gerais, Brazil, fernando@dcc.ufmg.br.

2475-1421/2021/10-ART144
https://doi.org/10.1145/3485521

inconvenient for large-scale applications. Moreover, there are scenarios in which the acquisition of execution traces from representative inputs is impractical, such as for end-user mobile applications, for instance. In the absence of profile information, binary optimizers are likely to be of little benefit, as their guided optimizations have been shown to be quite sensitive to the quality of the data available [Panchenko et al. 2019, 2021; Wade et al. 2017].

Paper Contributions. In this paper, we argue that the aforementioned drawbacks of binary optimizers can be mitigated by coupling them with static profilers. A static profiler [Wu and Larus 1994] aims to approximate the data produced by its dynamic counterpart. To accomplish this goal, the static profiler relies on static program characteristics to infer said program's run-time behavior. To support our thesis, we use, as a starting point, the static profiler proposed by Calder *et al.* [Calder et al. 1997]—a technique called *Evidence-Based Static Prediction* (ESP). Although elegant and effective, Calder et al.'s static analyses cannot be directly applied to binary optimizers. As originally conceived, ESP only predicts the direction of two-way conditional branches. However, modern binary optimizers need the execution probabilities (or frequencies) of these branches.

To overcome this limitation, we adapt ESP to estimate how frequently each branch will be executed. We call this adaptation VESPA — short for *Vintage ESP Amended*. VESPA consists of training and prediction phases. During training, we collect an assortment of static features from programs. By observing the execution of a corpus of representative benchmarks, we associate these features with the probability that a branch is taken. During prediction, we build a static profile for an unknown program. To this end, the model predicts the outcome of this program's branches based on the features that characterize them. These probabilities are then passed to a binary optimizer to guide its optimization decisions. The gains that we have observed in this paper stem from three contributions, which we summarize as follows:

Features: we show that the original program features adopted by Calder et al. [1997] would not be sufficiently informative to be used in the context of binary optimization, because they were designed to be collected from a compiler's intermediate representation. From this observation, we have modified ESP's feature set, removing some of them, adapting others to be applicable at the binary level, and adding new program characteristics. Section 3.2 discusses these adaptations.

Model: similarly to the program features, we show that Calder et al. [1997]'s original decision tree and neural network models are still amenable to improvements. In particular, the neural network greatly benefits from changes to its architecture. Additionally, modeling the problem as a regression task rather than a classification one also improves results. This contribution is the subject of Section 3.5.

Engineering: the implementation of a static profiler in the context of a binary optimizer requires a number of non-trivial engineering decisions which we summarize in Section 3. Although not a scientific contribution, we believe that such decisions deserve mention, at least to prevent others from facing some of the dead-ends that we have encountered. Thus, we explain how we map probabilities to branch frequencies, how we deal with partially available control flow graphs, and how we have extracted program features from binary code.

Summary of Results. We have implemented our static profiler on top of the BOLT binary optimizer [Panchenko et al. 2019]. This new version of BOLT has been evaluated on four large executables: clang, GCC, MySQL and PostgreSQL. Experiments reported in Section 5 show that the binaries generated by BOLT fed with the static profiles inferred by VESPA are on average around 6% faster than the same program compiled with clang -O3. This number is still far from what can be accomplished with dynamic profiling information (a performance improvement of almost 35.0%

on average). However, three observations are in order: first, VESPA does not require any form of dynamic profiling, thus simplifying the use of binary optimizers. Second, a performance improvement of 6% on top of clang (version 12.0) -O3 is not to be overlooked—at that optimization level, clang applies 281 passes to the input program. Furthermore, clang already uses, by default, Ball and Larus [1993]'s heuristics to lay out basic blocks—a well-established static profiling technique. Third, our static profiler outperforms previous well-established branch prediction heuristics, including Wu and Larus's, by considerable margins.

2 CODE PLACEMENT

We call *spatial locality* a measure of the distance between the virtual memory addresses of two consecutively executed instructions of a program. Instructions that are located at close memory addresses are said to have *high spatial locality*. If instructions tend to be executed together in time (for instance, if the execution of an instruction succeeds the execution of another), then it is desirable that they have *high spatial locality*. In this way, they are likely to be fetched in the same cache line, via one single memory access. There are different ways to fulfill this desire. For instance, the code of functions that form a caller-callee relation can be laid out next to each other [Ottoni and Maher 2017]. Yet, the most common technique to optimize spatial locality is to place *basic blocks* that tend to be executed in sequence on successive memory addresses.

A program is formed by basic blocks, which are arranged in a control flow graph (CFG). A basic block is a sequence of non-branch instructions, except for the last one; thus, if we disregard preemption at the OS level, these instructions will be executed in sequence. The instructions in a basic block are naturally placed together in memory. A CFG is a directed graph where each vertex is a basic block. An edge between two blocks, e.g. BB_i to BB_j, indicates that control may flow from BB_i to BB_j. In this case, we say that BB_j is a *successor* of BB_i. The need to place instructions likely to be executed together in close memory addresses leads to a problem that we shall call the *Basic-Block Placement Problem*, which we define as follows:

Definition 2.1 (The Basic Block Placement Problem (BBPP)). **Input:** a Control Flow Graph $G = (V_b, E_b)$, whose vertices V_b are basic blocks, and whose edges E_b determine possible program flows; plus a map F that associate edges in E_b with *execution frequencies*. The execution frequency of a CFG edge is a positive integer that estimates how often that edge will be traversed during program execution. **Output:** an ordering (also called a *Linearization*) L of V_b that minimizes the execution cost of G. The cost of traversing an edge $BB_i \rightarrow BB_j$ is zero if BB_j immediately follows BB_i in L; otherwise, it is $F(BB_i \rightarrow BB_j)$.

Definition 2.1 is a rather simplistic description of basic block placement. It disregards the relative distance between basic blocks by adopting a null-or-full cost model. The ultimate measure of efficiency of any solution to BBPP is execution speed, and that is the success criterion that we shall adopt in Section 5. Nevertheless, the simplification adopted in Definition 2.1 still allows us to provide the reader with a clear overview of basic block placement, as Example 2.2 illustrates.

Example 2.2. Figure 1 shows an instance of the basic block placement problem. Estimates of execution frequency are given as numbers in black boxes. Section 2.1 explains different techniques to calculate such estimates. Figures 1(b) and 1(c) show two different linearizations of the program in Figure 1 (a). The first yields a cost of $1 + 2 + 98 + 100 = 201$; the latter, a cost of $1 + 2 + 2 + 100 = 105$. Therefore, the second is preferable over the first. However, that is not even the best linearization for the program in Figure 1. We leave it for the interested reader the task of finding the best solution.

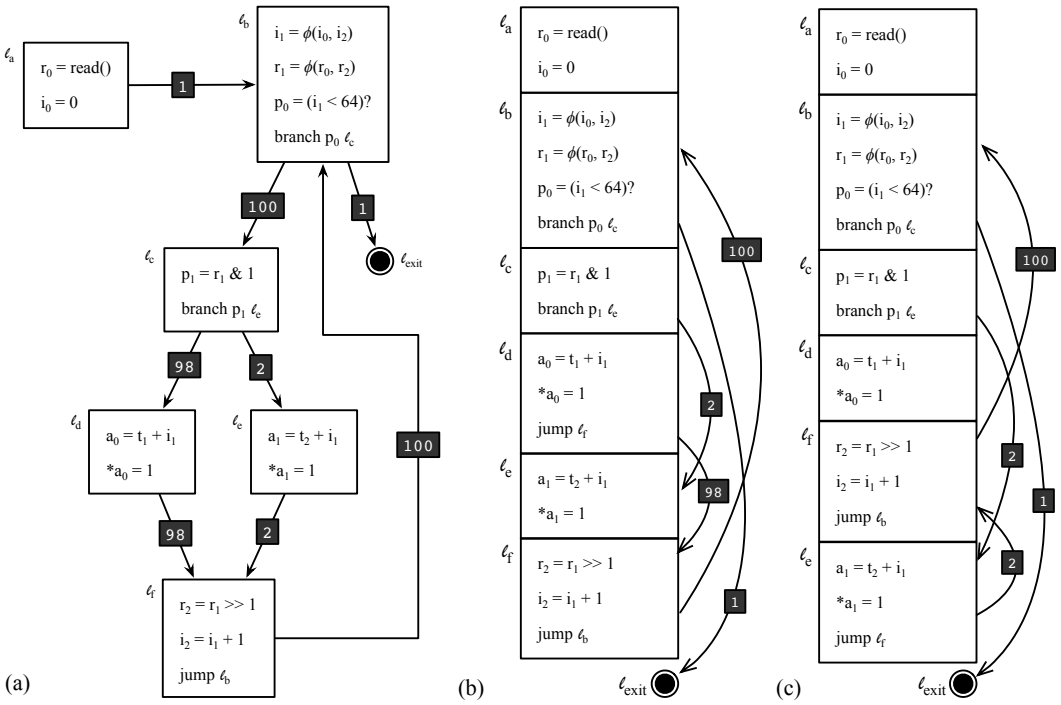

Fig. 1. (a) An instance of the basic block placement problem. (b) A solution with cost 201. (c) A solution with cost 105. The lower the cost of the linearization, the more efficient it is, according to that cost model.

2.1 Program Profiling

Different authors might use the word "profiling" with the most diverse semantics. Because this notion plays a fundamental role in our presentation, we shall restrict its meaning. Therefore, to avoid confusion, throughout this paper, whenever we use the term profiling, we shall be referring to the sense introduced in Definition 2.3. Notice that this understanding amounts to finding the estimate F of execution frequencies, previously mentioned in Definition 2.1.

Definition 2.3 (Program Profiling). A program profiler is any technique that associates the edges of a program's CFG with *execution frequencies*. The notion of execution frequency is introduced in Definition 2.1. The map F of CFG edges to frequencies is called a *profile*.

There are two ways to build program profilers: dynamically or statically. We discuss former in Section 2.1.1, and the latter in Section 2.1.2. Notice that both methodologies lead to approximate solutions to BBPP. Dynamic profilers determine the most-likely successor of a basic block based on known program inputs, which might not be representative of other inputs. Static techniques, in turn, stumble on Rice's Theorem [Rice 1953]. It follows as a corollary of said theorem that, given a basic block with multiple successors, it is undecidable to determine, statically, which of them will be executed.

2.1.1 Dynamic Profiling. A dynamic profiler is a tool that observes the execution of a program, and retrieves useful information from these observations. In the context of this paper, we assume that a dynamic profiler builds a table mapping each of the program's control flow branches to information regarding its runtime behavior. This information can be, for instance, the number of

times the branch runs, or the likelihood that the branch is taken. There exist different techniques to create such a record, based on either code instrumentation or sampling.

Instrumentation consists in recompiling the target program, augmenting it with additional code to maintain a log of branch executions, usually by associating a counter to each branch. While it provides extremely precise data, instrumentation has two major drawbacks. First, it complicates the build process of the target application, because it requires an additional step to inject instrumentation code. Second, instrumentation tends to significantly impact the program's memory and runtime performance. Overheads depend on the type of profiling information that is acquired, but can be as high as 100x [Rimsa et al. 2021, 2019]. Due to these limitations, instrumentation-based techniques are usually not chosen for use in industry-quality tools, such as `gprof` or BOLT.

The second dynamic profiling technique is *sample-based*. In this case, the program's execution is monitored selectively. Selective observation means that only a subset of the program's instructions are inspected. This approach minimizes regressions in the execution time of programs. Sampling is usually performed at the hardware level, via performance counters. While less precise than instrumentation-based profilers, sampling techniques still provide reasonably accurate results at much lower cost, with the added benefit of not requiring recompilation. Therefore, mainstream profilers tend to be sampling-based.

2.1.2 Static Program Profiling. A *static profiler*, like its dynamic counterpart, builds a map that associates control flow edges with their execution frequencies. The vast majority of static techniques depart from *branch probabilities*. In other words, they seek to estimate the chance that a conditional branch can be taken. Standard algorithms are then used to map such estimates into frequencies [Wu and Larus 1994]. The input of a static profiler is only a program's code; thus, contrary to a dynamic technique, it has no access to that program's inputs. Therefore, a static profiler must calculate the chance that a given branch is taken based only on the syntax—and its implied semantics—of the program where that branch exists.

The simplest static profiling heuristic that we are aware was introduced in the 1980's. This approach is called the *Backward-Taken/Forward-Not-Taken* (BTFNT) guess [Smith 1981]. The BTFNT heuristic is based on the observation that branches that continue loop execution (backward edges) tend to be taken. This heuristic is quite effective, considering how simple it is, as we shall demonstrate in Section 5. Still in the 1980's, further improvements have been proposed on top of BTFNT, like considering the branch's opcode [Smith 1981], or more cases that might cause the termination of loops [Bandyopadhyay et al. 1987].

Nevertheless, in spite of the progress made in the 1980's, the golden decade of static branch prediction was the 1990's. Ball and Larus [1993] have introduced a set of tests that are applied—in order—onto a branch to predict its outcome. This approach is still used today, for instance, in the LLVM compiler, to solve the Basic Block Placement Problem. Although simple and elegant, Ball and Larus's first match approach suffers from an obvious shortcoming: sometimes more than one heuristic applies to the same branch. However, only the first matching candidate is used in this case. To circumvent this limitation, Wu and Larus [1994] introduced the concept of evidence combination. To this effect, they use Dempster-Shafer Theory [Dempster 1967] to blend multiple heuristics.

Still in the 1990's, Calder et al. [1997] showed how to use machine-learning techniques to predict the outcome of branches statically. Calder et al. used decision trees and neural networks to predict the direction of two-way conditional branches. In other words, they were answering a binary question about the direction of a branch. Calder et al. observed that neural networks and decision trees had similar performance, at least in their experimental setup. Therefore, one might favour the use of decision trees given their relative simplicity when compared to neural networks. A few years later, Desmet et al. [2005] extended Calder et al.'s work with additional syntactic features.

They have further reiterated that decision trees are good models for branch prediction. Our work departs from that point; hence, porting Calder et al.'s and Desmet et al. [2005]'s contributions to the context of a binary optimizer.

3 THE VESPA STATIC PROFILER

VESPA is a form of static profiling designed to guide binary optimizations. VESPA removes the need for dynamic profiling to enable binary optimizations, thus simplifying and broadening the applicability of binary-optimization tools. Although VESPA is a general static profiler that can be used with different binary optimizers, in this work we study and evaluate VESPA's use along with the BOLT binary optimizer [Panchenko et al. 2019].

For the sake of completeness, Figure 2 provides an overview of how BOLT is typically used. BOLT takes as input a linked binary program, which has been produced by a standard compiler toolchain. Notice that BOLT does not require access to the input program's source code. In addition to the input binary, BOLT also takes a file containing profile data as input. This profile data is typically obtained through sampling-based hardware performance counters (e.g. using the Linux `perf` tool). This profile data is then transformed into BOLT's input format through an adaptation phase (using the `perf2bolt` tool). With both the input binary and the properly formatted profile data, BOLT then performs a series of optimizations to the binary. In practice, the most effective optimizations applied by BOLT are basic-block and function reordering, which greatly reduce CPU front-end stalls [Panchenko et al. 2019].

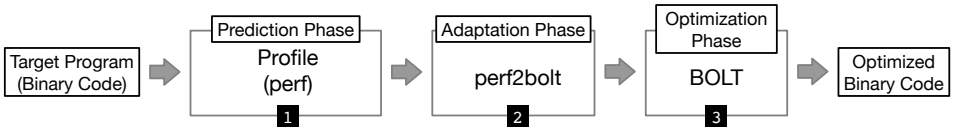

Fig. 2. The typical BOLT pipeline.

3.1 VESPA in a Nutshell

Our static profiler eliminates stage 1 in Figure 2. VESPA replaces this step with an approximation of the input program's runtime behavior created based on observing the execution of a collection of training programs. Figure 3 illustrates VESPA's approach. VESPA's usage can be divided into two parts. The first, "Training", consists in building the model that approximates the behavior of programs. The second, "Prediction" consists in applying the knowledge acquired in the training stage onto unseen programs, emulating, via a static profile, the dynamic profile that BOLT would use. The rest of this section provides more details about each of these phases, and their internal steps. But, before diving into that, we present Example 3.1 to give a quick overview of VESPA's operation, with focus on the prediction phase.

Example 3.1. Figure 4 illustrates how prediction works for a given branch in the target program. Prediction starts with feature extraction. Features are mined from the target program syntax. In this example, we assume a set of four features (whose descriptions appear in Table 1). Three of them assume binary values; the other—`CMP_OPCODE`—ranges over a category of values. Once features are extracted, they are arranged into a *feature vector*, which Figure 4(c) shows. Said vector is passed to a "`prediction`" function. The value that results from applying this function onto the feature vector is the probability that a branch is taken. Figure 4(d) shows a very simple linear predictor. The goal of the training phase is to build an accurate prediction function. The models that we explore in Section 3.5 are non-linear.

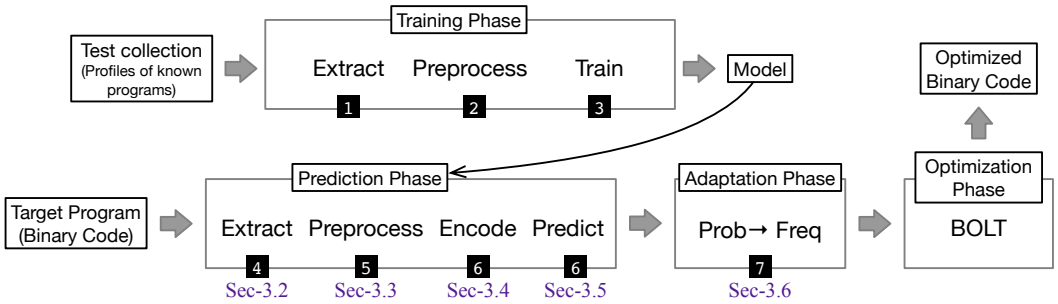

Fig. 3. VESPA's pipeline.

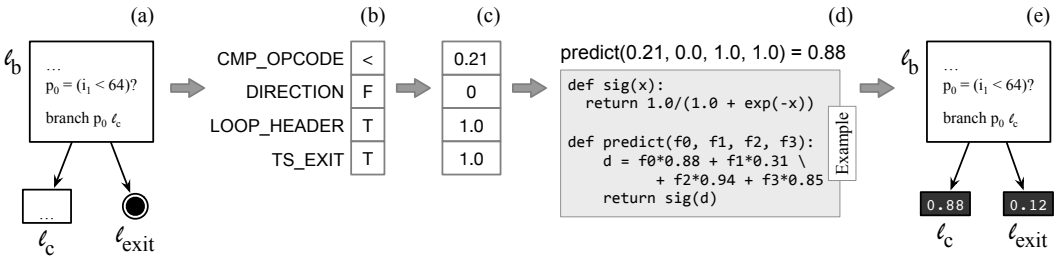

Fig. 4. (a) Code snippet from the program seen in Figure 1. (b) Feature extraction. (c) The feature vector. (d) A very simple mock-up predictor. (e) Branching probabilities.

3.2 Feature Extraction

In the context of this paper, predictions are carried out by matching out *static program features* with expected behaviors. In our case, the "expected behavior" of a conditional branch is determined by profiling a known set of programs. Following Pereira et al. [2018], we use Definition 3.2 to delimit the notion of program feature that is relevant to this presentation.

Definition 3.2 (Static Program Feature). Given a program I, and a branch instruction ι, a static program feature $f(I, \iota)$ is any characteristic of I, related to ι, with the following properties:

- [Finite]: $f(I, \iota) \in S$, where S is a finite set;
- [Static]: $f(I, \iota)$ depends only on the syntax of I;
- [Consistent]: if $f(I, \iota) = x$, then x is unique;
- [Polynomial]: $f(I, \iota)$ can be computed in polynomial time.

Example 3.3 (Static Program Feature). Consider the instruction "branch p_0 ℓ_{exit}" in Figure 1(a). Several program features are associated with this instruction. A first feature is the opcode used to implement this instruction, e.g., in x86's machine code: `jne, je, jg, jle, jl`, and `jge`. A second feature is the opcode used to produce the value p_0. This opcode could be any arithmetic or logic operation. A third feature is the direction of the branch: forward or backward. In this example, we have a forward branch.

On The Choice of Program Features. The original work of Calder et al. defined 30 static program features. We could reuse 21 of them, which are listed in Table 1. The remaining features were left out for two reasons. First, the feature called Language refers to the programming languages used by Calder, namely C or Fortran. This information is innocuous in our domain: the low-level binary

Table 1. Categorical branch features proposed by Calder et al., and reused in this work.

ID	Feature	Description
1	OPCODE	the opcode of the branch instruction
2	CMP_OPCODE	the opcode of the predicate operation
3	FS_END_OPCODE	the opcode of the terminal instruction in the fallthrough successor basic block
4	TS_END_OPCODE	the opcode of the terminal instruction in the taken successor basic block
5	DIRECTION	whether the branch is backward or forward (i.e. if it flows to an earlier or later address in the program's layout)
6	LOOP_HEADER	whether the basic block containing the branch is a loop header
7	PROCEDURE_TYPE	whether the procedure containing the basic block encompassing the branch is leaf, non-leaf or recursive
8	OPERAND_RA_TYPE	the opcode of the left hand side operand
9	OPERAND_RB_TYPE	the opcode of the right hand side operand
10	TS_DOMINATES	whether the taken successor basic block is dominated by the basic block containing the branch
11	TS_POSTDOMINATES	whether the taken successor basic block post-dominates the basic block containing the branch
12	TS_LOOP_HEADER	whether the taken successor basic block is a loop header
13	TS_BACKEDGE	whether the taken successor basic block is a back edge
14	TS_EXIT	whether the taken successor basic block contains a call to exit the program
15	TS_CALL	whether the taken successor basic block contains a procedure call
16	FS_DOMINATES	whether the fallthrough successor basic block is dominated by the basic block containing the branch
17	FS_POSTDOMINATES	whether the fallthrough successor basic block post-dominates the basic block containing the branch
18	FS_LOOP_HEADER	whether the fallthrough successor basic block is a loop header
19	FS_BACKEDGE	whether the fallthrough successor basic block is a back edge
20	FS_EXIT	whether the fallthrough successor basic block contains a call to exit the program
21	FS_CALL	whether the fallthrough successor basic block contains a procedure call

code of x86. Second, eight features were left out because BOLT's intermediate representation did not provide the necessary information to implement them.

While we have dropped some features proposed by Calder et al., we have also included a few new features in VESPA. Tables 2, 3 and 4 show the new set of features that we use to extend Calder et al.'s work. The motivation for these inclusions is empirical, meaning that we have evaluated several different features, which were, in the end, not chosen to compose the final feature set. Features like the proportion or the absolute number of instructions of different kinds (shifts, multiplications, stores, comparisons, etc) in the 'then' or 'else' target of a branch did not reduce the prediction error observed in training and in testing, for instance. Rather, they were leading to overfitting. We have evaluated features based on the following factors:

(1) The feature was originally proposed by Calder et al. [1997].
(2) The feature was proposed by Namolaru et al. [2010]. This work includes, for instance, the number of load and store instructions in the targets of branches.
(3) The feature emerged during discussions with the original engineers that worked in the BOLT project. The DELTA_TAKEN feature, for instance, came out of these discussions.
(4) The feature was thought out by the authors of this paper.

Table 2. First set of additional numeric branch features proposed by this work.

ID	Feature	Description
22	NUM_STORES	number of store instructions in the basic block containing the branch
23	NUM_LOADS	number of load instructions in the basic block containing the branch
24	NUM_CALLS	number of call instructions in the basic block containing the branch
25	NUM_CALLS_INVOKE	number of invocation instructions in the basic block containing the branch
26	BASIC_BLOCK_SIZE	number of instructions in the basic block containing the branch
27	NUM_BASIC_BLOCKS	number of basic blocks in the function containing the branch
28	DELTA_TAKEN	the absolute difference between the address of the branch and its taken basic block address
29	NUM_OUTER_LOOPS	number of outer loops in the function containing the branch
30	TOTAL_LOOPS	number of loops in the function containing the branch
31	LOOP_NUM_EXIT_EDGES	number of loop exit edges in the function containing the branch
32	LOOP_NUM_EXIT_BLOCKS	number of basic blocks which are the destination of an edge leaving a loop in the function containing the branch
33	LOOP_NUM_EXITING_BLOCKS	number of basic blocks which exit a loop in the function containing the branch
34	LOOP_NUM_LATCHES	number of loop latch basic blocks in the function containing the branch
35	LOOP_NUM_BLOCKS	number of basic blocks in the innermost loop within the function containing the branch
36	LOOP_NUM_BACKEDGES	number of backedges in the innermost loop within the function containing the branch
37	NUM_INDIRECT_CALLS	number of indirect call instructions in the function containing the branch
38	NUM_SELF_CALLS	number of recursive calls in the function containing the branch

The criteria used to keep a feature were:

(1) It was easy to extract from the program's binary representation.
(2) It could reduce prediction error in at least one of the benchmarks available in the test set.
(3) We could explain, at least intuitively, why the feature would lead to certain branch results.

Some of our new features try to measure the "semantic weight" of the destination of a branch. This metric measures how heavy on different types of instructions is the basic block that is the target of a branch. Features 22-26, 52, 39-50, in Tables 2 and 3, estimate this weight. A second category of new features refers to the function that contains the branch. These features include characteristics of the function, such as the number of outermost loops that it contains (29), or the maximum depth of any loop nest (51). Finally, features 53-55 encode structural characteristics of the basic block that contains the branch. For instance, Feature 53 determines if the basic block that contains the branch is the exit point of some loop. Section 5 analyzes the effects of our new set of features compared to the assortment originally proposed by Calder et al.. That said, our feature set is not definitive: we still believe that it is possible to refine it further.

Table 3. Second set of additional numeric branch features proposed by this work.

ID	Feature	Description
39	TS_NUM_LOADS	number of memory load instructions inside the taken successor basic block
40	TS_NUM_STORES	number of memory store instructions inside the taken successor basic block
41	TS_BASIC_BLOCK_SIZE	number of instructions in the taken successor basic block
42	TS_NUM_CALLS	number of call instructions in the taken successor basic block
43	TS_NUM_INDIRECT_CALL	number of indirect procedure calls in the taken successor basic block
44	TS_NUM_CALLS_INVOKE	number of invoke instructions in the taken successor basic block
45	FS_NUM_LOADS	number of memory load instructions inside the fallthrough successor basic block
46	FS_NUM_STORES	number of memory store instructions inside the fallthrough successor basic block
47	FS_BASIC_BLOCK_SIZE	number of instructions in the fallthrough successor basic block
48	FS_NUM_CALLS	number of call instructions in the fallthrough successor basic block
49	FS_NUM_INDIRECT_CALL	number of indirect procedure calls in the fallthrough successor basic block
50	FS_NUM_CALLS_INVOKE	number of invoke instructions in the fallthrough successor basic block

Table 4. Third set of additional branch features proposed by this work. These features are all categorical.

ID	Feature	Description
51	MAXIMUM_LOOP_DEPTH	maximum loop depth in the function containing the branch
52	LOOP_DEPTH	depth of the innermost loop encompassing the branch
53	LOCAL_EXITING_BLOCK	whether the basic block containing the branch is a loop exiting block
54	LOCAL_LATCH_BLOCK	whether the basic block containing the branch is a loop latch block
55	LOCAL_LOOP_HEADER	whether the basic block containing the branch is a header for an innermost loop in the function
56	FUN_TYPE	whether the function containing the branch is classified as simple or not by BOLT

3.3 Feature Preprocessing

Once the features have been collected, it is necessary to clean the training/prediction data, and make it suitable to be fed to a Machine Learning model. This preprocessing happens in several steps. The first step consists of dealing with incomplete data. There may be branches for which there is insufficient data available in the disassembled binary to determine their static properties. Usual culprits for incompleteness are indirect jumps. If the disassembler fails to determine successors for indirect jumps, then VESPA will be unable to compute features such as those related to dominance and post-dominance, for instance. Similarly, due to the layout of the disassembled binary, in some cases it may be difficult to track which instruction computes the predicate that controls a branch. We perform a best effort to identify as many patterns as possible to find predicates, but these do not cover all of them. Therefore, some branches features related to instruction operands (such as OPERAND_RA_TYPE and OPERAND_RB_TYPE) or predicates (CMP_OPCODE) may be missing.

Incompleteness occurs for a small amount of branches. In other words, our current implementation of a feature miner computes features for about 95% of all the instructions in our dataset. In face of incomplete information, the missing values are replaced by default values. These defaults are the values most likely to occur in practice. However, a few missing features cause us to remove the branch altogether. Examples of these properties include features related to dominance and post-dominance data.

3.4 Feature Encoding

Encoding is the process of mapping the features related to a given branch into a numeric *feature vector*. This process is non-trivial because features, although finite (see Definition 3.2), do not all share the same type. Depending on these types, features can be classified according to Definition 3.4.

Definition 3.4 (Feature Classification). Given a program I and a branch $\iota \in I$, a static program feature $f(I, \iota) = s, s \in S$ is either numerical or categorical. $f(I, \iota)$ is numerical if S is a totally ordered set. In this paper, every numerical feature ranges over $S = \mathbb{N}$. The feature is categorical if S is any finite, countable, albeit unordered set.

Example 3.5. Features NUM_STORES and NUM_LOADS in Table 2 are numerical. Features like TS_DOMINATES and TS_EXIT are categorical. These features are booleans. Indeed, most of the categorical features range on $S = \{true, false\}$. However, there are categorical features that range over larger sets, like OPCODE and CMP_OPCODE, in Table 1.

Encoding Numerical Features. To encode numerical features we apply standardization, or mean removal and variance scaling; hence, ensuring that the numeric values have a mean of 0 and a standard deviation of 1.0. Thus, for each numerical feature we subtract the mean of its distribution from its value and divide the resulting number by the feature's standard deviation. Standardization ensures that all features are treated equally, in regards to its range of values. We also apply batch normalization for the DNN model, to accelerate its learning during the training phase [Laurent et al. 2016].

Handling Categorical Features. To handle categorical data, we use two different encoding approaches: for values with few categories, we use *one-hot encoding*. In one-hot encoding, the original feature is removed, and replaced by one binary feature for each of its possible values. The PROCEDURE_TYPE feature, for example, can take one of three values ('leaf', 'non-leaf' or 'call-self'). Once encoded, it is replaced by three binary features, such as $\texttt{IS\_LEAF} \in [0, 1]$. One-hot encoding is limited to features with up to three categories to keep feature vectors short. For features ranging over more than three categories (OPCODE, CMP_OPCODE, TS_END_OPCODE,FS_END_OPCODE, LOOP_DEPTH and MAXIMUM_LOOP_DEPTH), we use either ordinal label encoding (when the features are to be used in the Decision Tree model) or embedding (when using a Neural Network). Ordinal encoding maps each category to a unique integer. Embeddings are similar to one-hot encoding, in that they map a single categorical feature to a numerical vector. However, while one-hot encoding maps a feature of n possible categories to a vector of n *binary* values, embeddings map categories to vectors of *continuous* values. The specific values assigned to each vector to represent a category are then learned during the training process of the neural network itself.

3.5 The Machine Learning Models

This work is heavily based on Calder et al.'s static branch predictor. However, this paper uses a very different machine-learning apparatus. Such departure from Calder et al.'s work is motivated for two reasons. First, that model would *classify* branches as either taken or not-taken, whereas we need a *regression model*. In other words, we need a model that associates a branch with a probability

that it will be taken. From this probability, we can use previous work [Wu and Larus 1994] to compute the static profiler, i.e., an estimate of how often the branch will be traversed. More about this conversion will be discussed in Section 3.6. Second, Calder et al. have experimented with two different machine-learning models: neural networks and decision trees. Our attempt to use the latter did not lead to effective results, as we will explain soon, and our implementation of the former follows a very different architecture than the model initially proposed by Calder et al..

The Neural Network Model. Figure 5 describes the architecture of the neural network adopted in this work. A neural network contains a sequence of layers: an initial input layer, a final output layer, and one or more intermediate layers, called *hidden layers*. When a neural network has more than one hidden layer, it is typically called a deep neural network (DNN)—such is the case of the model seen in Figure 5. We use a fully connected DNN with five dense layers. The dense layer computes a linear function of its inputs, then applies a non-linear activation function on the result of this linear combination. For the non-linear operator, we use Rectified Linear Unit (ReLU) [Apicella et al. 2021]. ReLU is employed in every intermediate hidden layer, with a dropout strategy with probability 0.5. However, for the last hidden layer, we use a sigmoid activation function, and a dropout probability of 0.2. The sigmoid function is necessary at this stage, for its application maps whichever results are produced by the neural network into a probability, i.e., a number between 0 and 1.0.

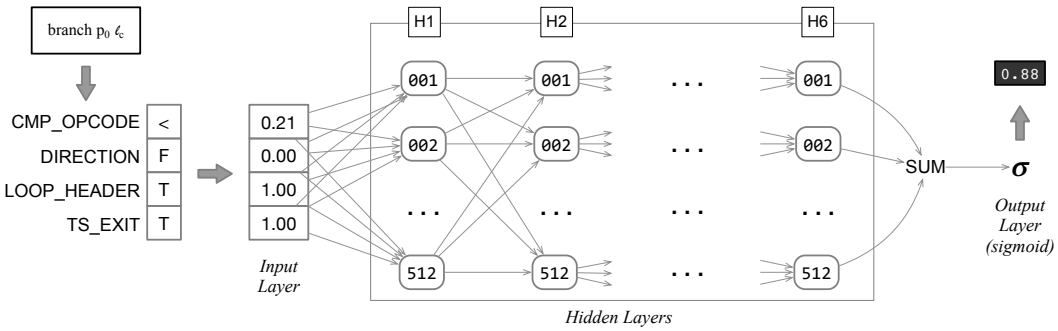

Fig. 5. VESPA's Deep Neural Network architecture.

Each node of the network has a weight and bias associated with it, which are its learnable parameters. The goal of the training phase of the network is to learn these parameters. The more accurate is this learning phase, the more effective are the predictions produced by the neural network. Thus, the combination of those learnable parameters and the activation function will determine the success of the probability predictor.

Dealing with Low-Frequency Branches. The data used to tune the learnable parameters of the neural network are the profiles of known programs. This data is very irregular: some branches are exercised billions of times, while others are executed only once. Due to this imbalance, some form of normalization is in order. Calder et al. dealt with this issue by replicating static branch samples so their frequencies would reflect their normalized dynamic weight in the program. Additionally, we have used the normalized branch weight of each static branch as a sample weight when fitting to the training set.

Experience with Other Models. In addition to using a neural network, Calder et al. also proposes to use a decision tree to predict branches. We tried to adapt this model; however, we met with two problems. First, the straightforward use of the decision tree with all our additional features

was not practical: the model ends up consuming an excessively large amount of memory. Our attempts to use a random forest required more than 4GB to build the regressor. Second, pruning techniques resulted in models that were not producing statistically significant results. We evaluate these results in Section 5.1. Nevertheless, we speculate that it is still possible to use decision trees as a means to generate a static profiler; however, the engineering of such a model is not a trivial endeavor, and we leave it as future work.

Feature Selection. While having a large feature set may allow a model to better learn branch patterns, having a feature space of high dimensionality can significantly impact its training time. Thus, we perform feature selection to reduce the number of features we use for training. To this end, we first leverage the feature ranking from the decision tree building algorithm. When building a decision tree, the algorithm ranks features by importance, and splits nodes based on the most important features. We initially reduced our feature set to the 30 most relevant features in this ranking. We then further reduce this set by performing recursive feature elimination, by fitting a model and then eliminating the feature which least affects the model's performance. We eventually settled on a subset of 26 features, 17 of which come from Calder et al.'s original set, while the remaining 9 are amongst our new proposed features.

3.6 From Probabilities To Execution Frequencies

Following Definition 2.3, the goal of a static profiler is to infer a map F that associates edges of the program's control flow graph with *estimates* of execution frequencies. Good estimates tend to approximate, at least in terms of proportions, the average runtime behavior of the program when fed with actual inputs. However, although execution frequencies are the end goal of a static profiler in the context of this paper, not every previous implementation of such technique delivers this mapping. For instance, Calder et al. [1997] and Ball and Larus [1993]'s techniques produce *Branch Predictions*, whereas the first phase of Wu and Larus [1994]'s analysis produces *Branch Probabilities*. These two notions are relevant to this presentation. The former—predictions—because that will be our starting point, given that we reuse much of Calder et al.'s techniques. The latter—probabilities—because, like Wu and Larus's, our inference algorithm also requires them as intermediate results, before arriving at frequencies. For the sake of completeness, we define these two concepts below:

Definition 3.6 (Branch Predictions and Probabilities). A *Branch Prediction* is an answer to the following question: given a conditional branch, which of its paths is the most likely to execute? A *Branch Probability* is an estimate of how likely a given path will be taken.

Example 3.7. Figure 6(a) shows branch predictions produced by an *oracle* for the program in Figure 1(a). An oracle is an optimal predictor. In other words, given an execution of the program, it predicts as taken the most traversed paths of a program. Although an oracle cannot be used in practice as a mechanism of static inference (for it requires running the program), it lets us compare the accuracy of different predictors: the closer to the oracle a predictor is, the more accurate—for that particular execution of the program—it is. Figure 6(b) shows the execution probabilities that our static profiler would infer for the program in Figure 1(a). The way such inference is performed is the subject of Section 3.5. Finally, Figure 6(c) shows the execution frequencies produced for that program, using the techniques originally proposed by Wu and Larus [1994].

Mapping Probabilities into Frequencies. The regression models discussed in Section 3.5 produce probabilities. However, BOLT performs optimizations based on absolute branch counts (execution frequencies), which is the information provided by `perf`. To map probabilities to frequencies, we resort to Wu and Larus [1994]'s method. Wu and Larus describe intraprocedural and interprocedural algorithms to perform this conversion. We have reimplemented these two algorithms within BOLT.

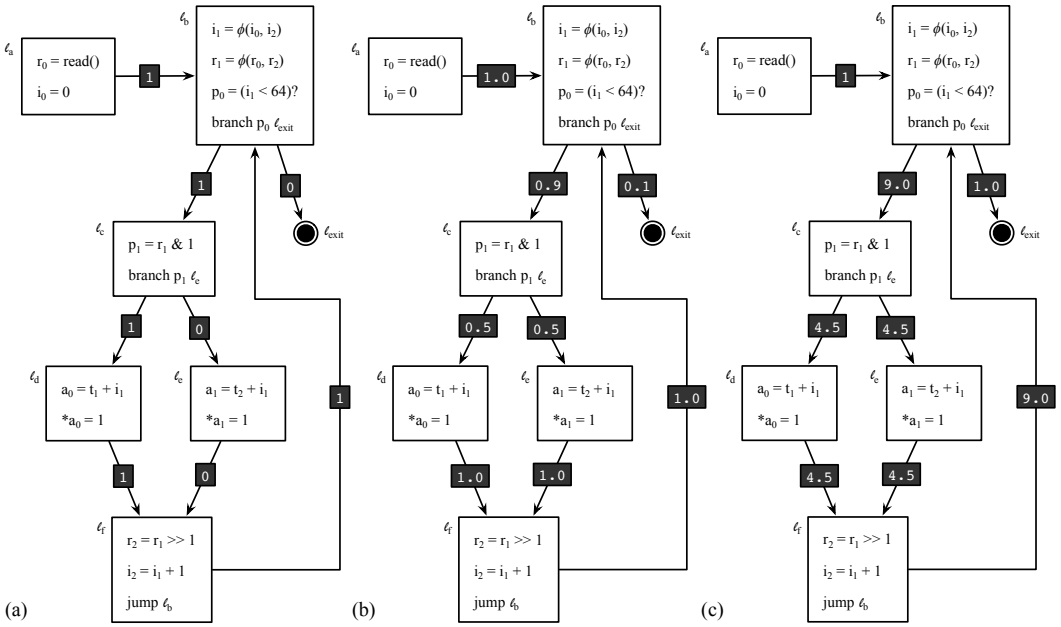

Fig. 6. (a) Branch prediction for the program in Figure 1(a). (b) Estimate of execution probabilities for the same program. (c) Estimate of execution frequencies.

Porting Wu and Larus's techniques to BOLT required a few engineering expedients. First, those algorithms cannot handle irreducible control flow graphs. Yet, such CFGs occur in practice. When such is the case, the Law of Flows cannot be met (the frequencies of incoming edges must equal the frequencies of outgoing edges). Thus, for the sake of practicality, given these inherent imprecisions, validation allows for a deviation of up to 20% between incoming and outgoing frequencies. Indirect branches poses another problem. We use static analysis to reconstruct CFGs from binary code. Due to indirect branches, the CFG thus produced might contain *false positives*: edges that will never be traversed in practice. Typical examples are switches implemented with jump tables. In this case, we split frequencies equally among all the edges. Finally, because Wu and Larus's inference produces floating point numbers, but BOLT requires integers, we decided to simply multiply every result by 1,000,000. The choice for this value is empirical: it is large enough to convert into discrete numbers almost every edge frequency, and is small enough to not cause integer overflows. If overflows happen, then the frequency of an edge is set to the maximum integer in the target machine.

Engineering. For implementing VESPA's step to convert probabilities into execution frequencies, we basically had two options: either implement it as an external adaptation tool, or implement it directly within BOLT. We opted for the latter because Wu and Larus's techniques require building the CFGs of the input program, a capability that BOLT already had. Therefore, we implemented this step as an alternative execution mode in BOLT, which can be enabled by a command-line option. When this option is used, BOLT attempts to read the input profile data from a `.pdata` probabilities file, rather than the usual `.fdata` file produced by `perf2bolt`. Moreover, we also added two extra flags to BOLT to control whether to convert probabilities into absolute counts using the interprocedural or intraprocedural algorithms proposed by Wu and Larus.

4 INCORPORATING PREVIOUS WORK INTO BOLT

In addition to the evidence-based static profiler that constitutes the main contribution of this paper, we have augmented BOLT with classic static profiling heuristics. Such additions were natural, because all the necessary equipment to read branch frequencies computed statically were in place to support our machine-learning model. Table 5 enumerates the other static profiling heuristics added to BOLT. These techniques run independently. In other words, their results cannot be combined: users of our version of BOLT must choose which static profiler they will employ.

Trivial Branch Predictors. We call a *trivial predictor* any technique that guesses the outcome of a branch without the support of any kind of evidence and runs in constant time per branch. We have added five such predictors to our implementation of BOLT. Table 5 list them. They have been originally evaluated by Smith [1981]. These techniques assume that a branch is taken with certain probability. Probabilities considered in this work are 0% (**Never taken**), 20%, 50% (**Unbiased**), 80% and 100% (**Always taken**). Section 5 evaluates three predictors: never taken, unbiased and always taken. We omit the other two, as they have not brought any improvement upon these three trivial predictors. We do not intend that any of these five static profilers be used in practice. Rather, we employ them as a mechanism to test a null-hypothesis, namely: trivial prediction cannot outperform the more elaborate heuristics proposed in the literature.

Table 5. Choice of heuristics for static profiles in BOLT.

Flag	Description
-heuristics-based=always	trivially predicts 100% probability for the taken successor
-heuristics-based=never	trivially predicts 100% probability for the fallthrough successor
-heuristics-based=weakly-taken	trivially predicts a 20%/80% probability split for the taken/fallthrough successors, respectively
-heuristics-based=weakly-not-taken	trivially predicts an 80%/20% probability split for the taken/fallthrough successors, respectively
-heuristics-based=unbiased	trivially predicts a 50% probability for all edges
-heuristics-based=wularus	uses the heuristics described in Ball and Larus' work [Ball and Larus 1993] in combination with Dempster-Shafer theory as described by Wu and Larus [Wu and Larus 1994]

Classic Predictors. The other branch prediction heuristic that we have added to BOLT is the static profiler proposed by Wu and Larus [1994]. As mentioned in Section 1, Wu and Larus use Dempster-Shafer Theory [Dempster 1967] to combine nine branch-characterization features proposed by Ball and Larus [1993]. Table 6 lists these features. This table also shows the probability that a branch with the given feature is taken. We have computed such probabilities over a training corpus yet to be described in Section 5. In that section, we shall refer to this heuristic as **Wu-Larus**.

Notice that, by using LLVM to generate the baseline executable programs, we are already testing the original work of Ball and Larus [1993]. LLVM uses a variation of that paper to layout the basic blocks that compose a program. The original implementation of Ball and Larus has suffered small modifications to be incorporated into LLVM due to undocumented observations made by compiler engineers over the years. As a consequence, two features proposed by Ball and Larus have been dropped, and the relative importance between them has been modified[1].

[1]The interested reader can find more about LLVM path profiler through the work of Preuss [2010]. The heuristics that LLVM 12.0 uses to compute the probabilities that branches are taken are implemented in https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/BranchProbabilityInfo.cpp.

Table 6. List of heuristics as described by Wu and Larus [1994].

Heuristic	Description	Taken Prob
Loop branch heuristic (LBH)	Predict edges back to a loop's head as taken. Predict edges exiting a loop as not taken.	88%
Pointer heuristic (PH)	Predict that a pointer comparison against null or against another pointer will fail.	60%
Opcode heuristic (OH)	Predict that comparisons of an integer for less than zero, less than or equal to zero, or to a constant will fail.	78%
Guard heuristic (GH)	If a comparison where a register is an operand is used before being defined in a successor block, where the successor block does not post-dominate the comparison's block, predict that the comparison branch to the successor block.	84%
Loop exit heuristic (LEH)	Predict that a comparison in a loop in which no successor is a loop head will not exit the loop.	80%
Loop header heuristic (LHH)	Predict that a successor block which is a loop header or pre-header, and does not post-dominate the branch's basic block, will be taken.	72%
Call heuristic (CH)	Predict that a successor basic block which contains a function call and does not post-dominate the branch's basic block will not be reached.	55%
Store heuristic (SH)	Predict that a successor basic block which contains a store instruction and does not post-dominate the branch's basic block will not be reached.	75%
Return heuristic (RH)	Predict that a successor containing a return instruction will not be reached.	62%

5 EVALUATION

In this paper, we explore five research questions, namely:

- **RQ1–Accuracy**: What is the accuracy of the model we propose in this paper, compared to the model proposed by Calder et al.?
- **RQ2–Performance**: What are the performance gains of our approach when compared to other ways to add (static or dynamic) profiling information to binary optimizers?
- **RQ3–Correlation**: How do the different profiling approaches impact the locality of instructions in the ICache, and how does this impact influence program performance?
- **RQ4–Training**: How long does it take for VESPA to generate a trained predictive model?
- **RQ5–Application**: What is the optimization time that VESPA requires, and how does it compare to the time taken by other versions of BOLT?

Experimental Setup: Experiments were executed on a dedicated server featuring an Intel Xeon E5-2620 CPU at 2.00GHz, with 16GB RAM, running Linux Ubuntu 18.04, with kernel version 4.15.0-123. All binaries used for training and as baseline were compiled using clang 12 (at -O3). Our profiler was built on top of BOLT's publicly available implementation[2]. Execution statistics were reported with Linux's perf, version 4.15.18.

Benchmarks Used in the Development of the Model: To train our model, we used the eleven programs in SPEC CINT2006 plus 226 programs from the LLVM test-suite, for which we collected execution profiles using instrumentation. We have used 80% of the branches from these programs to train the prediction model. Of the remaining branches, half were used to test the model and the

[2]Specifically, commit 8028b7b.

other half to validate it (for tuning purposes). Therefore, we have employed 243 programs in the development of the branch prediction model. This corpus gave us 2,093,873 two-way conditional branches. However, only 513,316 branches were associated with profile information. Thus, this is the corpus used to train the prediction model.

Benchmarks Used in the Validation of the Model: For validation, we used the four largest open-source programs that were available to us. Size, in this case, is ordered based on the length of the text segment of the final executable program. The chosen benchmarks are:

- The **clang** compiler, version 7, as used in BOLT's reference tutorial [Panchenko 2018], compiling its own source code as input;
- The **GCC** compiler, version 7, compiling clang 7's source code as input.
- The **MySQL** database management system, version 8.0, with the `oltp_point_select` benchmark from the SysBench[3] suite as input.
- The **PostgreSQL** database management system, version 13.2, with the `select-only` benchmark from the PgBench[4] suite as input.

The requirement of a large binary with a large portion of hot code is of capital importance for validation, because BOLT's optimization effects can only be perceived once the hot parts of a program are too large to fit into the instruction cache. If the program has a large binary footprint, but its hot part is small, then it will fit into the instruction cache. And, in this case, the benefits of the extra locality achieved by BOLT cannot be noticed. That is the reason why we have not used benchmarks from neither the LLVM test suite, nor from SPEC CPU for validation. As mentioned in BOLT's reference tutorial [Panchenko 2018], a good rule of thumb is that if an application has over 10 misses per thousand instructions, then it is a good indication that it will be improved by BOLT.

On Dynamically Linked Libraries: The benchmarks use dynamic shared libraries; however, BOLT does not optimize dynamically linked code sections. Therefore, `libc` and other shared libraries cannot be the source of any performance variation that we shall report in this section. Alvares et al. [2021] classifies as *visible* the portion of the program that the compiler can optimize, i.e., its source code, and *invisible* the ensemble of instructions coming from external libraries. They have shown that most of the instructions processed when the programs in SPEC CPU2017 run with reference inputs are visible. Following this classification, once we run our benchmarks with the available inputs, we also notice that most of the instructions processed are visible. As an example, 76.21% of the instructions in GCC, when optimized with BOLT's dynamic profiler, are visible. Once we use VESPA's static profiler, this number increases to 78.42%. This growth reflects the fact that the dynamic profiler is more efficient in optimizing the visible part of the program.

Experimental Methodology: Running time numbers reported in this section are the average of ten executions. All of our runtime experiments had P-values under 0.05 with confidence level of 95% when testing for the null hypothesis via Student's T-Test. The null hypothesis would indicate lack of performance variation due to binary optimization with either static or dynamic profiling.

The Competing Approaches: Results that we report in this section are relative to clang -O3. We use this baseline to compare nine different versions of BOLT, which we enumerate below:

(1) **Dynamic profile**: This is the official distribution of BOLT, which uses dynamic profiling information to lay basic blocks out.
(2) **Limited dynamic profile**: This approach is similar to the previous one, except that it does not include profiling data for indirect branches. The purpose of this limited version of BOLT is to show how much of its optimization potential is hindered by the lack of indirect branch

[3]Available at https://github.com/akopytov/sysbench.
[4]See https://www.postgresql.org/docs/10/pgbench.html for further information about PgBench.

data. Thus, this approach provides a better estimate of VESPA's potential, which currently does not add profiling information to indirect branches.

(3) **VESPA**: The contribution of this paper—a static profiler to guide code placement decisions.
(4) **Calder**: This version is and approximation of Calder et al.'s original work, which produces branch frequencies instead of binary predictions. Some features originally proposed by Calder et al. [1997] are only meaningful in a high-level programming language like C; hence, they could not be extracted from the binary representation used by BOLT.
(5) **Wu-Larus**: This is the approach proposed by Wu and Larus [1994], which combines multiple prediction heuristics via Dempster-Shafer's formulae [Dempster 1967].
(6) **Unbiased**: This approach assigns, for every branch, a 50% chance of being taken.
(7) **No profile**: In this case, BOLT does not use any profiling information. In the absence of profiling information, BOLT maintains the initial basic block placement. However, due to three optimizations: stripping no-ops, tail call removal and branch reversal, BOLT can still speedup programs.
(8) **Never taken**: This version of BOLT assumes that every conditional branch always evaluates to false; hence, it is never taken.
(9) **Always taken**: This is another trivial prediction heuristic, that assumes that every branch evaluates to true; hence, it is always taken.

5.1 RQ1–Accuracy

VESPA builds on previous work on static profiling due to Calder et al.. Our approach, however, adds significant changes to the modeling of the problem and the techniques used to perform predictions. In this section, we explore how our additions improve the performance of the static profiler.

Methodology: We set out to compare the prediction performance of VESPA against the ESP technique proposed by Calder et al.. As mentioned in Section 3, however, we decided to use regression models rather than the original classifiers, which makes the performance of the models hard to compare conceptually. Nevertheless, we felt the additional program features we propose could already provide significant benefits when compared to the original ESP technique. Therefore, we trained two versions of the Decision Tree classifier proposed in the original paper: one using Calder et al.'s static features, and another using our extended set of program features. We then compared their branch prediction performance on the test set of programs. Since VESPA uses regression, we also trained two versions of a Decision Tree regressor with the same sets of features. We then compare the predictive performance of these predictor pairs. Finally, we evaluated the accuracy of the Deep Neural Network regressor which VESPA ultimately uses as its predictor.

Discussion: Figure 7 summarizes the result of this experiment with Decision Tree classifiers. Each graph shows the Precision-Recall curve of its respective model, with metrics shown for a varying threshold probability to separate branches between the taken/not taken classes. These curves are used to evaluate the overall quality of a given classification model, considering the trade-off between true positives and its positive predictive value, over a large variety of values for its threshold parameter. A "perfect" classifier would have a curve formed by two straight lines at the top and right of the graph, indicating 100% Precision and 100% Recall for every threshold. The closer a classifier's curve comes to that, the better. The performance of a classifier can be quantified via its Average Precision (AP), a metric that indicates the classifier's average precision across all thresholds. As shown in the figure, the original ESP model achieves an average precision of 0.71, while the VESPA Decision Tree has a 0.80 average. This result indicates that our extended feature set significantly improves the model's prediction performance, even when using only the original model proposed by Calder et al. [1997].

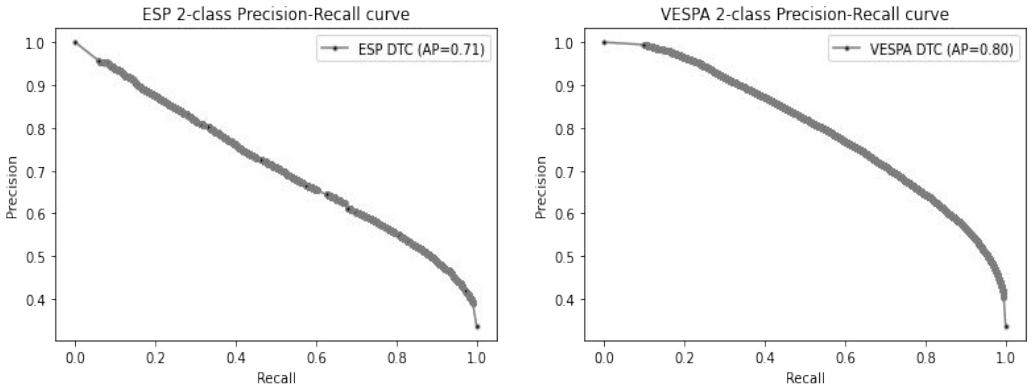

Fig. 7. Comparison of predictive performance between Calder et al.'s ESP and VESPA.

On the Choice of Regression Model. The results shown in the rest of this section have been obtained through a neural network. However, we have also experimented with other models. In particular, following Calder et al.'s recommendation, we have tested the accuracy and effectiveness of decision trees. Yet, contrary to Calder et al.'s work, we had to adapt the decision tree to perform regression, instead of classification. Therefore, the output of these models must be branch probabilities rather than the binary taken/not taken answer.

We trained two versions of a Decision-Tree based regressor: one using Calder et al.'s original features, and another using our proposed feature set. Their performances have been evaluated on the four binaries that we use for validation. A metric typically used in the evaluation of regression models is the distance (error) between the observed and the predicted values. One such metric is the Root-Mean-Square Error (RMSE), which is the square root of the average of squared errors. The values used to compute this metric is the error measured in comparison with the probability of each branch being taken during dynamic execution. The lower is the error, the more accurate is the static predictor, as compared to a dynamic profiler. Table 7 summarizes the performance of these regressors. The regressor trained with VESPA's feature set achieves lower prediction errors for all binaries in our evaluation set. This result shows that improvements due to our selection of features remain positive in decision trees.

Table 7. Root-Mean-Squared error of Decision Tree regression models.

	ESP	VESPA (Decision Tree)
clang	0.24	0.22
GCC	0.23	0.21
MySQL	0.28	0.26
PostgreSQL	0.25	0.22

Experiments performed with Decision Trees are useful to show that our extended feature set improves prediction. However, as mentioned in Section 3, we found neural networks more accurate, while also being easier to implement. Therefore, we settled on a Deep Neural Network regressor as our final predictor, trained with our proposed feature set. Table 8 summarizes the performance of this regressor. For every single binary, the DNN regressor has a lower error than the aforementioned Decision Trees.

Table 8. Root-Mean-Squared error of VESPA's DNN regressor.

	VESPA (DNN)
clang	0.18
GCC	0.18
MySQL	0.23
PostgreSQL	0.21

5.2 RQ2–Performance Improvements

In this section, we evaluate the performance improvements of the different binaries produced by either the baseline compiler or by BOLT when fed with different kinds of profiling information (either static or dynamic).

Methodology: The procedure used to collect dynamic profile information to be used by BOLT for each binary varied. In the case of clang and GCC, we used the steps described in BOLT's documentation, i.e. each compiler bootstraps itself. For these benchmarks, the performance improvements are in regards to execution time speedup. For MySQL, we used a combined profile containing merged information from running several of the benchmarks in the SysBench suite. We follow a similar procedure for PostgreSQL, merging profiles from several benchmarks of the PgBench suite. For these benchmarks, the performance improvements metrics are in regards to throughput, in other words, the number of Transactions Per Seconds (TPS) that each Database Management System (DBMS) is capable of processing.

Discussion: Figure 8 shows the relative performance improvements between different versions of each binary versus its baseline version. As expected, the binary optimized by BOLT with a fully dynamic profile provides by far the best results, with an overall performance improvement of 34.46% (geometric mean) over clang -O3. However, once dynamic profiling information for indirect jumps are removed, this benefit falls to 19.13%. Because none of the static profiles that we use in this section deals with indirect branches, we believe that this is a better point of comparison. The binary optimized using the VESPA-generated profile yields the best results among the static profiling heuristics. VESPA achieves approximately 9% of performance improvements on clang and MySQL. For GCC and PostgreSQL, the gains are more modest, albeit noticeable. When used without any profiling information, BOLT delivers performance improvements of about 2% onto the majority of the benchmarks tested. This performance improvement is due to a series of automatic code optimizations that BOLT performs by default, which include removing no-ops, reversing branches and replacing recursive calls with loops whenever possible. Our approximation of Calder et al.'s implementation does not fare well on the benchmarks used in Figure 8. It achieves less than 1% of performance improvement on GCC, being worse than Wu and Larus's static profiler.

The execution frequencies inferred statically enhance BOLT's ability to generate efficient code by a statistically significant margin. In every case, VESPA's p-values were less than 0.01. In this experimental setup, optimizations enabled by Wu and Larus's heuristics, while profitable, fare no better than a trivial profile that attributes a 50/50 probability to every branch. We believe that this disappointing result is due to the fact that clang already employs a simplified version of Wu and Larus's heuristics to layout basic blocks. Finally, trivial profiles that attribute a single direction to every branch impact negatively the quality of the code produced by BOLT in all benchmarks.

5.3 RQ3–Correlation

Panchenko et al. explain that most of the performance improvements delivered by BOLT comes out of its ability to increase locality in the instruction cache (I-cache). This section evaluates how

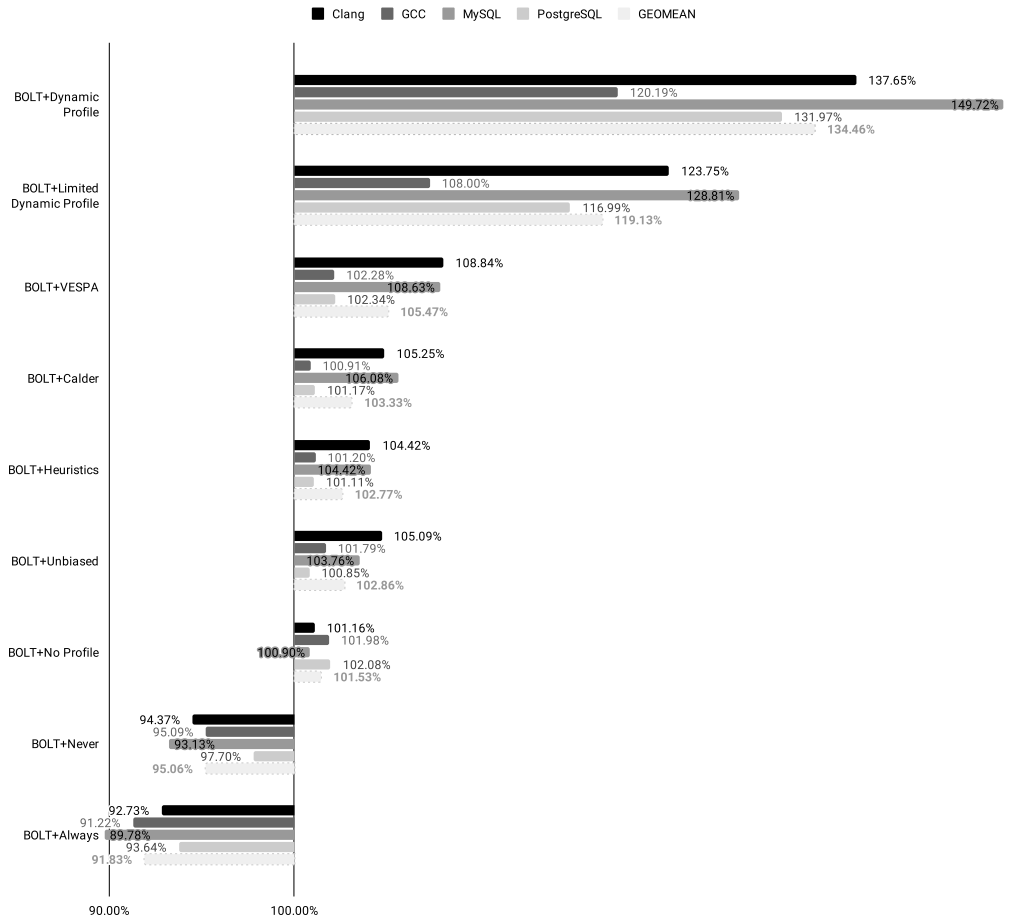

Fig. 8. Performance improvements of different versions of the benchmarks' binary vs. its baseline built with clang -O3.

the different profiling techniques impact locality in the I-cache. A strong correlation between performance and locality will provide further indication that our gains are due to actual changes in BOLT's code alignment algorithm.

Methodology: We followed the same procedures described in Section 5.2 to generate and execute optimized binaries. The I-cache miss rate of each benchmark is gauged via Linux's `perf`. As already explained in Section 5.2, we adopt two different metrics to measure performance variation, depending on the benchmark. For clang and GCC, performance is runtime speedup. However, for MySQL and PostgreSQL, performance is throughput measured in transactions per second. This difference is due to the implementation of the publicly available harnesses distributed with each benchmark.

Discussion: Figure 9 shows the improvement in I-cache misses for each binary. BOLT equipped with a fully dynamic profiler delivers the most significant results, achieving a reduction of 36.48%

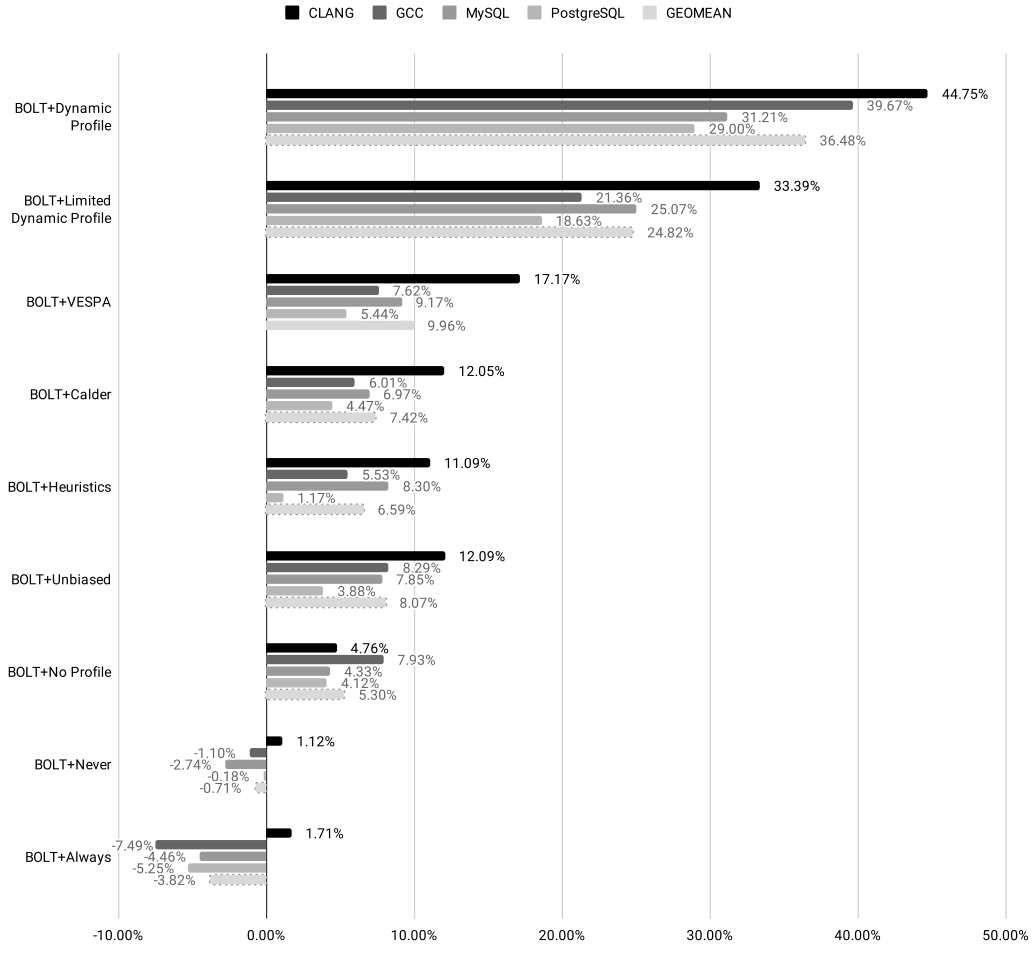

Fig. 9. Reduction in L1 instruction cache misses for different versions of the benchmarks' binaries.

in I-cache misses on average (geomean). VESPA, in turn, outperforms the other static approaches, obtaining a 9.96% geomean reduction in cache misses across the benchmarks.

Table 9. Correlation coefficients between reductions in L1 I-cache misses and performance improvements.

	clang		GCC		MySQL		PostgreSQL	
	CC	P-Value	CC	P-Value	CC	P-Value	CC	P-Value
Pearson	0.995	0.000	0.993	0.000	0.982	0.000	0.991	0.000
Spearman	0.983	0.000	1.000	0.000	0.950	0.000	0.952	0.000
Kendall	0.943	0.001	1.000	0.000	0.886	0.001	0.923	0.001

Table 9 shows how performance variation correlates with variations in cache misses. We compute Pearson, Spearman and Kendall correlations for each benchmark. All of the correlation tests had P-values lower than 0.001. If a P-value is lower than 2e-8, then we report it as zero. For clang and GCC, correlations are very strong, with nearly all coefficients scoring above 0.95. For MySQL and PostgreSQL, the correlation coefficients are slightly lower, yet still significant. These results strengthen earlier observations by Panchenko et al., namely, that BOLT improves program performance because it can reduce I-cache misses. Figure 10 provides visual indication of the strong linear correlations reported in this section. Lines represents a theoretical one-to-one relation between improvement in performance and reduction in I-cache misses.

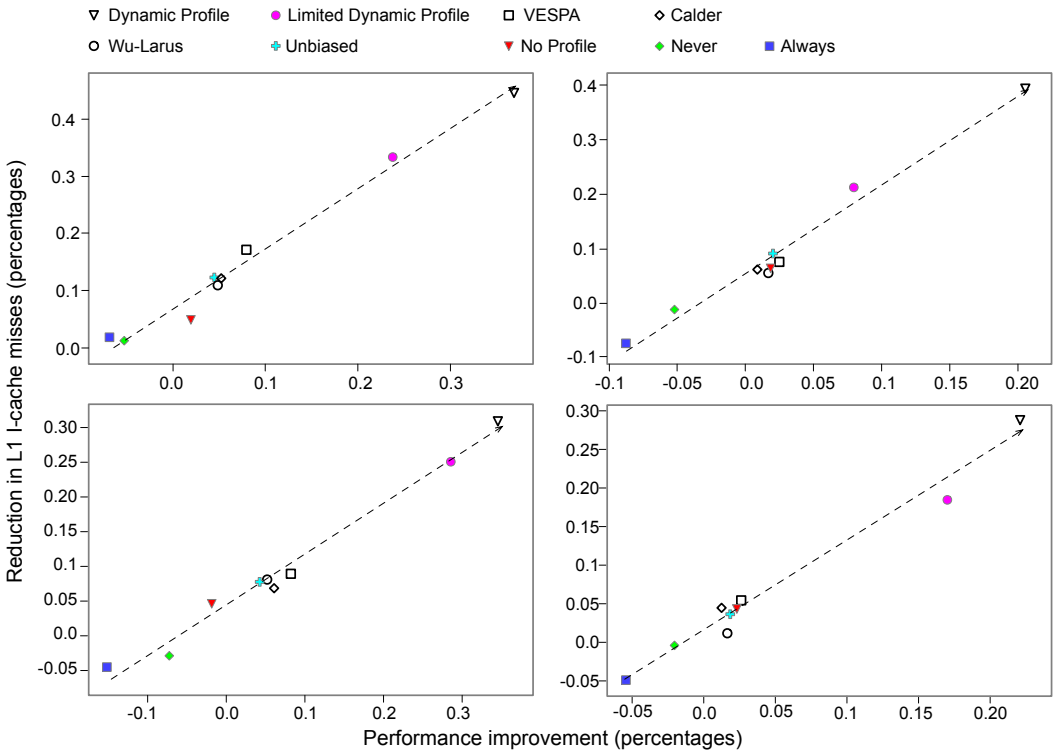

Fig. 10. Performance Improvements vs reduction in L1 I-cache misses for each benchmark compiled with different versions of BOLT.

5.4 RQ4–Training Time

While VESPA is able to create static profiles that approximate profiles created dynamically, its ability to do so relies on a preceding training stage performed on a large dataset of programs. This section reports how long this process takes in practice.

Methodology: We performed the training process described in Section 3, measuring the running time of each phase separately. This includes: instrumenting the binaries in the LLVM test suite and SPEC; running all the binaries, collecting dynamic profiles, and converting the profiles to BOLT's input format; extracting program features for branches in all programs; preprocessing and merging the static feature data; and finally encoding and training the deep neural network.

Discussion: The execution time of each of the phases in the training process were:

- Instrumentation: Using BOLT to generate instrumented version of each of the 237 programs in our training set takes **35.21 seconds**.
- Profiled execution: Executing each instrumented binary and collecting dynamic profiles takes **1,036 minutes**.
- Feature extraction: Running BOLT with the feature miner to collect static features for all the binaries takes **29.75 seconds**.
- Preprocessing and merging: Preprocessing all the static feature files and merging them takes **179.84 seconds**.
- Encoding and training: Encoding features and training the Deep Neural Network with the entire branch dataset takes **45.5 minutes**.

Therefore, the entire process to build a static branch dataset and to train a model capable of making predictions took, in our experimental setup 1,085 minutes—approximately 18 hours. The bulk of this time was spent running the binaries to collect dynamic profiles and training the neural network. These phases took 95.4% and 4.1% of the total execution time, respectively.

5.5 RQ5–Application Time

One of the benefits of static over dynamic profiling is the possibility of optimizing binaries without having to run them. This benefit translates into a shorter time to produce executable programs. To gauge the extent of this advantage, this section compares the time taken to optimize a binary following each version of BOLT that we use in this paper.

Methodology: We build an optimized binary for each of the evaluated programs, using all our seven versions of BOLT. When timing BOLT with the dynamic profiler, we count the time to run the target application to collect samples, plus the time to run BOLT to disassemble and optimize the binary. In the case of VESPA, we measured: (i) the time to run BOLT to disassemble and collect static features; (ii) the time to run the predictor to generate a static profile; and (iii) the time to run BOLT again, this time fed with the static profile information. Notice that we omit training time, because this process only happens once, after which the model can be reused as-is. For each other version of BOLT, we replace the time mentioned in step (ii), above, with the time to apply the particular heuristics that characterizes that version.

Discussion: Figure 11 shows the total time taken to generate binaries for each optimization process. As expected, using a dynamic profile tends to be the more expensive than using a VESPA-generated static profile, with clang taking roughly 3.3x longer to optimize. GCC takes around 1.5x as long. However, for the MySQL binary, BOLT with a dynamic profile is actually faster, by a factor of about 30%. This is due to the relatively short running time of MySQL's profiling inputs, which allows for fast sampling. In contrast, VESPA spends time not only running the predictor, but also importing them and embedding them onto the program's binary representation. Nevertheless, we emphasize that in cases where VESPA takes longer to generate a binary, this overhead is constant. Using a dynamic profile, however, can take an arbitrarily long amount of time, due to the application's dynamic behavior. In other words, the duration of VESPA's optimization process is more consistent.

The other static profiles evaluated in this section are much faster than either BOLT + VESPA or BOLT + dynamic profiling. That is to be expected, as these approaches only involve a single execution of BOLT, with little or no other program analyses included. Nevertheless, these results provide a good baseline to measure the overhead introduced by VESPA. When compared to a trivially generated static profiler, VESPA takes roughly 4.5x as long to generate an optimized binary. We emphasize again that this overhead is constant, as opposed to a dynamic profile, whose overhead depends on the running time of the target application.

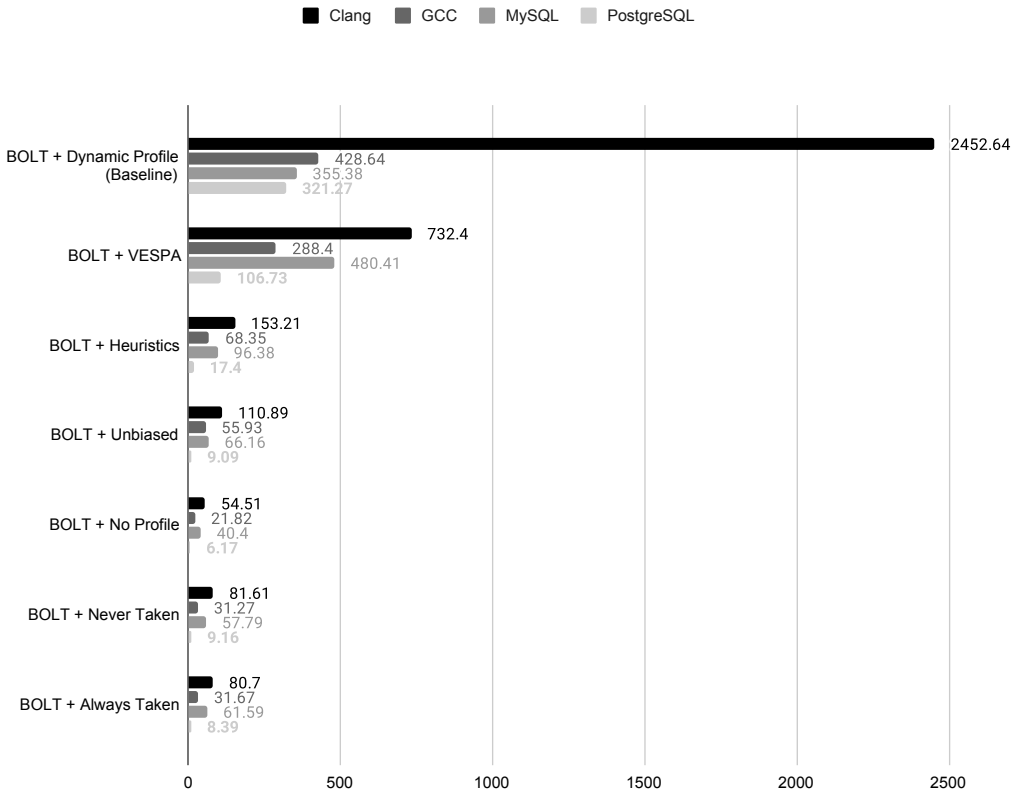

Fig. 11. Time used to generate binaries (in seconds). We omit the time used by our implementation of Calder et al. work, because it uses the same infrastructure as VESPA; hence, is similar.

6 RELATED WORK

The use of static profiling as an alternative to dynamic approaches in branch prediction has been proposed in several forms before. To the best of our knowledge, the first work to introduce the idea is due to Fisher and Freudenberger [1992]. Fisher and Freudenberger's contribution relies on the observation that branches vary little in direction, regardless of the program's input. Thus they propose that previous runs of a program can efficiently predict the direction of branches in future runs. Ball and Larus [1993] have introduced an alternative technique, which relies on a number of syntactic-based heuristics to predict the direction that branches will take. Wu and Larus [1994] further builds on this idea by determining a way to combine the evidence provided by multiple heuristics. This combination is performed via Dempster-Shafer's theory [Dempster 1967]. The technology discussed in this paper is similar to Wu and Larus's, in that we also combine multiple static aspects of a branch to predict its outcome. However, whereas Wu and Larus use nine features, we use 56, and whereas they combine heuristics via Dempster-Shafer formula, we use a regression model powered by a neural network.

6.1 Evidence Based Static Profiling

The work that most closely resembles ours, and on which we base our technique, is Calder et al. [1997]'s. Calder et al. argue that previous heuristic-based approaches are highly-specific to particular hardware architectures and programming languages—a fact that hinders their applicability. To mitigate this shortcoming, they introduce a learning-based method which relies only on static features of each branch, which they name Evidence-based Static branch Prediction (ESP). Nevertheless, the use of machine-learning techniques as an alternative to simpler heuristics in compiler optimization is also not a novel concept. Wang and O'Boyle [2018] provide a comprehensive survey on the application of different statistical models to guide compiler optimizations.

As mentioned in Section 1, VESPA is based on ESP's infrastructure, and it follows the typical workflow of a learning-based approach, with an initial training process preceding predictions. However, our approach differs and improves upon the original ESP in several aspects. Many of these differences emerged because we are revisiting Calder et al.'s work in the context of binary optimization. In this sense, this paper is the first to evaluate the impact of static profiling in this domain. The key differences to Calder et al.'s work are listed below. We claim them are original contributions of this paper:

Model: The original implementation of Calder et al. modeled the problem of predicting branches as a classification task, in which each branch's direction was determined as taken or not taken. Our implementation models the problem as a regression task, assigning to each branch a *probability* of being taken.

Features: We use a subset consisting of 17 of the 30 features originally proposed by Calder et al., which we have empirically found to be the most significant. Additionally, we use 9 new program features.

Architecture: As opposed to the Neural Network and Decision Tree models used by Calder et al., we have found that a Deep Neural Network architecture provides better prediction performance.

Frequency: While the machine-learning model outputs *probabilities* for each branch, BOLT relies on *execution counts* to carry out its optimizations. Therefore, we have extended BOLT with the capability to convert the branch probabilities into execution frequencies.

6.2 Hardware-Based Branch Prediction

There exists a vast literature concerning the prediction of branches at the hardware level. Indeed, branch prediction is one of the cornerstones of the fast execution of binary instructions in contemporary processors [Hennessy and Patterson 2011, Ch.3.3]. We emphasize that the techniques used in that domain—dynamic branch prediction—are fundamentally different from the approaches that we discuss in this paper. The key difference is the moment when predictions are performed. All the approaches mentioned in this paper, including BOLT guided by dynamic profiling, perform predictions offline; that is, before the program runs. Hardware-based methods, in turn, do it online, i.e., while the program is running.

Online techniques use dynamic information to carry out branch prediction. This information is stored in hardware-based tables such as the Local History Register (LHR), the Global History Register (GHR), and the Global Addresses Map (GA). This dynamic data can be used to feed regression and classification models. Such possibilities have been demonstrated by previous work [Kalla et al. 2017; Mao et al. 2018; Tarsa et al. 2019]. Nevertheless, due to being used at different moments; thus relying on different data, the online methodologies cannot be compared with our approach empirically.

7 CONCLUSION

This paper has discussed the application of a static profiler in the context of a binary optimizer. To this effect, we have revisited Calder et al.'s original work, and have modified it in several ways, so to achieve an implementation that could consistently outperform clang at its highest optimization level when applied onto large executables. Thus, the experiments discussed in Section 5 reveal that, although our static profiler still performs significantly worse than a dynamic profiler, it does deliver considerable performance improvements on top of highly optimized code. This claim is evidenced by a speedup of nearly 6% and a reduction of misses in the I-cache of nearly 10% on four benchmarks compiled with clang -O3.

Although our machine-learning model provides predictions that allow a certain degree of optimization, we believe that the model could be improved to better approximate dynamic profiles. One promising area for improvement is to have a more representative set of training programs, with a larger number of real-world applications rather than benchmarks. Another encouraging approach could be mitigating the inaccuracies introduced by the procedure of converting output probabilities to branch frequencies. For instance, implementing a block placement/code layout algorithm that relies on branch probabilities directly could possibly provide better performance, as it would not rely on assumptions implied in the probability-to-frequency conversion process.

Software. Our implementation is publicly available at https://github.com/angelica-moreira/BOLT. An artifact with scripts to reproduce our experiments is available in Docker Hub (`docker pull angelicamoreira/oopsla21artifact`) and in Zenodo (https://zenodo.org/record/5502310#.YT7Cty1h1QI, DOI `10.5281/zenodo.5502310`).

ACKNOWLEDGMENTS

The bulk of this work was developed by Angélica while visiting Facebook. During part of her PhD, Angélica was supported by scholarships from Facebook and CAPES. Fernando Pereira has been supported by CNPq (Grant 406377/2018-9); FAPEMIG (Grant PPM-00333-18) and CAPES (Edital CAPES PrInt). We thank Antony Courtney, Maksim Panchenko, Rafael Auler, Shaunak Kishore and other Facebook engineers for fruitful discussions. We thank José Wesley Magalhães, Luigi Soares, Vinícius Pacheco, Victor Campos and Andrei Álvares for reading a draft of this paper. Finally, we thank the OOPSLA reviewers for many comments and suggestions that greatly improved the final version of this work.

REFERENCES

Andrei Rimsa Alvares, Jose Nelson Amaral, and Fernando Magno Quintao Pereira. 2021. Instruction Visibility in SPEC CPU2017. *Journal of Computer Languages* 66 (2021), 1–10. https://doi.org/10.1016/j.cola.2021.101062

Andrea Apicella, Francesco Donnarumma, Francesco Isgrò, and Roberto Prevete. 2021. A survey on modern trainable activation functions. *Neural Networks* 138 (Jun 2021), 14–32. https://doi.org/10.1016/j.neunet.2021.01.026

Thomas Ball and James R. Larus. 1993. Branch Prediction for Free. *SIGPLAN Not.* 28, 6 (1993), 300–313. https://doi.org/10.1145/173262.155119

Sumit Bandyopadhyay, Vimal S. Begwani, and Robert B. Murray. 1987. Compiling for the CRISP Microprocessor. In *COMPCON*. IEEE Computer Society, San Francisco, California, USA, 96–101.

Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. 1997. Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Trans. Program. Lang. Syst.* 19, 1 (1997), 188–222. https://doi.org/10.1145/239912.239923

Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO*. Association for Computing Machinery, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

A. P. Dempster. 1967. Upper and Lower Probabilities Induced by a Multivalued Mapping. *Ann. Math. Statist.* 38, 2 (04 1967), 325–339. https://doi.org/10.1214/aoms/1177698950

Veerle Desmet, Lieven Eeckhout, and Koen De Bosschere. 2005. Using Decision Trees to Improve Program-Based and Profile-Based Static Branch Prediction. In *ACSAC*. Springer-Verlag, Berlin, Heidelberg, 336–352. https://doi.org/10.1007/11572961_27

Joseph A. Fisher and Stefan M. Freudenberger. 1992. Predicting Conditional Branch Directions from Previous Runs of a Program. In *ASPLOS* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/143365.143493

John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *PLDI*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

Bhargava Kalla, Nandakishore Santhi, Abdel-Hameed A. Badawy, Gopinath Chennupati, and Stephan J. Eidenbenz. 2017. A Probabilistic Monte Carlo Framework for Branch Prediction. In *CLUSTER*. IEEE Computer Society, New York, NY, USA, 651–652. https://doi.org/10.1109/CLUSTER.2017.29

C. Laurent, G. Pereyra, P. Brakel, Y. Zhang, and Y. Bengio. 2016. Batch normalized recurrent neural networks. In *ICASSP*. IEEE, Shanghai, China, 2657–2661. https://doi.org/10.1109/ICASSP.2016.7472159

David Xinliang Li, Raksit Ashok, and Robert Hundt. 2010. Lightweight Feedback-Directed Cross-Module Optimization. In *CGO*. ACM, New York, NY, USA, 53–61. https://doi.org/10.1145/1772954.1772964

Yonghua Mao, Junjie Shen, and Xiaolin Gui. 2018. A Study on Deep Belief Net for Branch Prediction. *Access* 6 (2018), 10779–10786. https://doi.org/10.1109/ACCESS.2017.2772334

Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *PLDI*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3192366.3192374

Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-Scale Data-Center Applications. In *CGO*. IEEE Press, United States, 233–244. https://doi.org/10.1109/CGO.2017.7863743

Maksim Panchenko. 2018. Optimizing Clang : A Practical Example of Applying BOLT. https://github.com/facebookincubator/BOLT/blob/master/docs/OptimizingClang.md, accessed on 2020-12-11.

Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *CGO*. IEEE Press, Washington, DC, USA, 2–14. https://doi.org/10.5555/3314872.3314876

Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: Powerful, Fast, and Scalable Binary Optimization. In *CC*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3446804.3446843

Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. 2018. Static Prediction of Silent Stores. *ACM Trans. Archit. Code Optim.* 15, 4, Article 44 (Nov. 2018), 26 pages. https://doi.org/10.1145/3280848

Adam Preuss. 2010. *Implementation of Path Profiling in the Low-Level Virtual-Machine (LLVM) Compiler Infrastructure*. Technical Report. University of Alberta. https://doi.org/10.7939/R3GF0MX64

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.1090/s0002-9947-1953-0053041-6

Andrei Rimsa, Jose Nelson Amaral, and Fernando Magno Quintao Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907

Andrei Rimsa, Jose Nelson Amaral, and Fernando Magno Quintao Pereira. 2019. Efficient and Precise Dynamic Construction of Control Flow Graphs. In *SBLP*. Association for Computing Machinery, New York, NY, USA, 19–26. https://doi.org/10.1145/3355378.3355383

James E. Smith. 1981. A Study of Branch Prediction Strategies. In *ISCA* (Minneapolis, Minnesota, USA). IEEE Computer Society Press, Washington, DC, USA, 135–148. https://doi.org/10.5555/800052.801871

Sriraman Tallam. 2019. *Profile Guided Optimizing Large Scale LLVM-based Relinker*. RFC. Google. https://github.com/google/llvm-propeller/blob/plo-dev/Propeller_RFC.pdf

Stephen J. Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham N. Chinya, and Hong Wang. 2019. Improving Branch Prediction By Modeling Global History with Convolutional Neural Networks. *CoRR* abs/1906.09889 (2019), 1–6. arXiv:1906.09889 http://arxiv.org/abs/1906.09889

April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *LCTES*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* PP (05 2018), 1–23. https://doi.org/10.1109/JPROC.2018.2817118

Youfeng Wu and James R. Larus. 1994. Static Branch Frequency and Program Profile Analysis. In *MICRO* (San Jose, California, USA). Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/192724.192725