UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Elisa Fröhlich

Static Projection of Profile Data Across the Optimization Pipeline

Belo Horizonte
2025

Elisa Fröhlich

Static Projection of Profile Data Across the Optimization Pipeline

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
2025

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Hydra: Static Detection of Hottest Blocks on Control Flow Graphs

ELISA FRÖHLICH

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Fernando M Q Pereira
PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]
Doutor MIRCEA TROFIN
Google

Sergey Pupyrev
Doutor SERGEY PUPYREV
Pesquisador Independente

Belo Horizonte, 12 de dezembro de 2025.

Acknowledgments

I would like to thank all my friends, old and new. Your companionship was essential in helping me persevere through these two years.

Among my old friends from Joinville, I begin by thanking Helena, who not only helped me discover who I am but continues to answer my questions to this day. I am deeply grateful to Miguel and Korhal, who listened to most of my complaints, specially about daily life in my apartment, and to Karla (who also heard many of those complaints), who visited me when I felt lonely toward the end of my master's program, which was very important for me. I also thank everyone with whom I shared some more "happy hours" during my visits to Joinville, especially Gio and Julia. I extend my gratitude to Professor Karina, my undergraduate project advisor, with whom I still share the best conversations and jokes about bureaucracy topics. Finally, I thank the members of the BRUTE competitive programming group, with whom I shared many discussions and Geoguessr game sessions over these two years.

Among my new friends from Belo Horizonte, I thank everyone in the Compilers Lab for every seminar, grading task force, and coffee break we shared, not only the official post-seminar or post-grading coffees, but also our informal coffees at some UFMG canteens (or at the physics department), from which so many interesting conversations arose. I am not naming anyone individually here because each of you was special to me in your own way, and I would be afraid of leaving someone out.

I thank my advisor, Fernando Pereira, who welcomed me not only into the Compilers Lab but also to UFMG. I am especially grateful for all the encouragement he gave me throughout the master's program, for introducing me to this fascinating research idea, and for the many insights that helped shape this project. I also thank Professor Cristiano, from UDESC, who connected me with Fernando two years ago.

I thank the members of the defense committee, Mircea Trofin and Sergey Pupyrev, for their valuable time and insightful comments during the review of this work.

I would like to thanks the members from the defense committee, Mircea Trofin and Sergey Pupyrev, for their valuable time and insights on the revision of this work.

I thank my parents, who have always encouraged me to keep focused in the studies, for all the support they give me.

This research was supported by grant APQ-00440-23 from the Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) and by a scholarship from Google. I am grateful for their support over the past two years, which made this research possible.

“*Here I am asking whether ‘you want to’ or ‘you don’t want to’, and instead you reply with whether you ‘can’ or ‘can’t’. That’s just grammatically incorrect, isn’t it?*”

(Inoue Kenji)

Resumo

Profile-guided optimization (PGO) é uma técnica bem estabelecida para melhorar o desempenho dos programas, sendo adotada pelos principais compiladores, como GCC, LLVM/Clang e o Microsoft Visual C++. Nela, dados de execução de um programa (perfil) são coletados para orientar o compilador na tomada de decisões mais eficientes, como inlining de funções críticas e organização de código. A coleta de perfil, no entanto, apresenta desafios: perfis estáticos podem não refletir o comportamento real do programa, enquanto perfis dinâmicos, obtidos por meio de execução instrumentada, têm alto custo computacional. Além disso, qualquer modificação no código-fonte — seja pelo programador ou pelas próprias otimizações do compilador — pode tornar o perfil coletado obsoleto, exigindo uma nova coleta. Para lidar com esse problema, este trabalho investiga a manutenção dos dados de perfil após uma sequência de otimizações, sem a necessidade de reexecutar o programa. Duas estratégias foram estudadas: o uso de perfis estáticos, e a projeção de perfis, que transfere informações de perfil de um programa original para sua versão transformada. Foram avaliadas diversas heurísticas para reconstruir os dados de perfil, incluindo uma abordagem baseada em um grande modelo de linguagem (LLM) com o GPT-4o e um método que compara, recursivamente, histogramas de operandos de blocos básicos e regiões de código para identificar similaridades estruturais. Os resultados demonstram que o método baseado em histogramas não só é mais simples, mas também supera em precisão as demais técnicas utilizadas em quase todos os cenários, incluindo as utilizadas pela LLVM e pelo otimizador de binários BOLT.

Palavras-chave: Análise Estática. Projeção de Profile.

Abstract

Profile-guided optimization (PGO) is a well-established technique for improving program performance, being integrated into major compilers such as GCC, LLVM/Clang, and Microsoft Visual C++. PGO collects information about a program's execution and uses it to guide optimizations such as inlining, and code layout. However, these very transformations alter the program's control flow, which may render the collected profiles stale or inaccurate. To deal with this problem, this work investigates how to maintain profile data after optimization without re-executing the program. We study two complementary strategies: prediction, which estimates likely hot code paths in the optimized program without any prior knowledge, and projection, which transfers profile information from the original control-flow graph to its transformed version. We evaluate several techniques for reconstructing profile data, including a large language model (LLM)–based approach using GPT-4o, and a lightweight method that compares opcode histograms of code regions recursively to identify structural similarities. Our results show that the histogram-based method is not only simpler but also consistently more accurate than both the LLM-based approach and prior prediction and projection techniques, including those implemented in LLVM and the BOLT binary optimizer.

Keywords: Static Analysis. Profile Projection.

List of Figures

2.1 (a) Function computing the sum of an arithmetic progression via iterative summation. (b) The respective function's CFG in SSA form. 19

2.2 Optimized versions of the function from Figure 2.1: (a) after GVN optimization, (b) after loop rotation optimization. 20

2.3 (a) Three executions producing profile data. (b) Control-flow graph of the program with profile information. (c) Optimized CFG after loop rotation, with profile data carried forward. 21

3.1 Static inference of profile information using the branch probability heuristics available in the LLVM compiler. Numbers associated with each block show the frequency of execution of that block. The pairs *prob*/*freq* associated with each edge show the probability that the edge is traversed and its execution frequency, i.e., how often the edge is expected to be traversed. 32

3.2 Prompt template used to request a profile prediction from GPT-4o. The template asks the model to identify the hottest basic block and to produce a ranking of blocks by hotness, along with reasoning. 33

4.1 (a) The program from Figure 2.3 (a), annotated with three hash variants for each basic block. (b) The same program after optimization with LLVM's Global Value Numbering. 38

4.2 (a) Matching of block regions using Algorithm 6 (`loop_matching`). We let G_{ref} be the left-side CFG, and G_{opt} the right-side one. (b) Matching of block regions after a recursive call of Algorithm 6. 43

4.3 Matching between the control-flow graphs of Figure 2.3 (c–d), illustrating how profile data are transferred from the reference code G_{ref} (left) to the optimized code G_{opt} (right). Dashed arrows indicate inferred correspondences between basic blocks and edges. Question marks mark missing or undefined profile values. 46

4.4 Result of applying Profi [23]'s reconstruction algorithm to the optimized program in Figure 4.3. The observed and reconstructed profiles are shown side by side, with adjusted frequencies ensuring flow conservation across the CFG. 47

4.5 Excerpts from the flow network creation process for the CFG in Figure 4.3, showing the network for (a) basic blocks bb_0 and bb_5, and (b) basic blocks bb_1 and bb_2. 50

4.6 Complete flow network without edges capacities and costs for the CFG and matched profile from Figure 4.3 (right). 51

4.7 Prompt template used to request a profile projection from GPT-4o. The template asks the model to identify the hottest basic block and to produce a ranking of blocks by hotness, along with reasoning, given an ordering of a previous version of the program. 62

5.1 Impact of different optimizations on the accuracy of classic profile projection heuristics. 72

List of Tables

5.1 Relative accuracy (Def. 5.1) of classic prediction heuristics. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks . . . 66
5.2 Relative accuracy (Def. 5.1) of classic profile projection heuristics. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks. . . . 68
5.3 Comparison between classic prediction heuristics and the LLM-based predictor. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks. . . . 69
5.4 Comparison between classic projection heuristics and LLM-based projection. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks. . . . 71
5.5 Execution time of classic profile prediction heuristics. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks . . . 74
5.6 Execution time of classic profile projection heuristics. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks. . . . 74
5.7 Execution time of classic profile projection heuristics per benchmark. Each row represents a different benchmark . . . 75

List of Abbreviations and Acronyms

CFG	*Control-Flow Graph*
PGO	*Profile-Guided Optimization*
IR	*Intermediate Representation*
SSA	*Static Single-Assignment*
GVN	*Global Value Numbering*
MST	*Minimum Spanning Tree*
PMU	*Performance Monitoring Units* ,
MCF	*Minimum-Cost Flow*
DAG	*Directed Acyclic Graph*
BMAT	*Paper from Wang et al. [54]*
SPM	*Paper from Ayupov et al. [2]*
Beetle	*Paper from Moreira et al. [39]*
Profi	*Paper from He et al. [23]*

List of Algorithms

1 Wu-Larus propagation algorithm, propagation subroutine. 30
2 Wu-Larus propagation algorithm. 31

3 Profile projection via block and edge matching. 35
4 Hash-based block matching. 37
5 Closest Histogram 41
6 Histogram-Based Loop Matching. 42
7 Histogram-Based Block Matching. 44
8 Filling missing profile information. 46
9 Create an Auxiliary Flow Network 48
10 Find Augmenting DAG in Flow Network 52
11 Run flow augmentation 53
12 Augmenting flow along DAG 54
13 Extracting weights from flow network. 55
14 Joining Isolated Components 56
15 Rebalance Undefined Subgraphs 56
16 Check if Edge is Considerable for Subgraph Finding 57
17 Find Undefined Subgraph 58
18 Subgraph Validity 59
19 Rebalance Subgraph 60
20 Check if Edge is Considerable for Subgraph Rebalancing 60
21 Rebalance Block 61

Contents

1 Introduction **14**

1.1 The Contribution of This Work 15

1.1.1 Software and Publications 17

2 Preliminary Definitions **18**

2.1 Control-Flow Graph 18

2.2 Program Profile 19

2.2.1 Instrumentation-Based Profiling 21

2.2.2 Sample-Based Profiling 22

2.3 The Hot-Order Problem 23

2.4 Static Profile Prediction 24

2.5 Static Profile Projection 24

3 Profile Prediction Heuristics **26**

3.1 Random Order 26

3.2 LLVM Predictor 27

3.3 Prediction with a Generative Model 31

4 Profile Projection Heuristics **34**

4.1 A General Projection Template 34

4.2 Hash-Based Matching 35

4.3 Histogram-Based Matching 36

4.3.1 Opcode Histogram Definitions 37

4.3.2 Matching Blocks by Similarity 39

4.3.2.1 The Closest Histogram Function 40

4.3.2.2 The Loop Matching Function 40

4.3.2.3 The Block Matching Function 43

4.4 Filling Out Missing Profile Information 45

4.4.1 Constructing an Auxiliary Network 47

4.4.2 Running Flow Augmentation 49

4.4.2.1 Finding an Augmenting DAG 50

4.4.2.2 Augmenting the Flow 51

4.4.3 Extracting Weights From Flow Network 52

4.4.4 Joining Isolated Components 54

4.4.5 Rebalancing Undefined Subgraphs 55
4.4.5.1 Finding Undefined Subgraph 57
4.4.5.2 Checking Subgraph Validity 58
4.4.5.3 Rebalancing Subgraph 59
4.5 Projection with a Generative Model 61

5 Experimental Evaluation **64**
5.1 RQ1 – Accuracy of Classic Profile Prediction Heuristics 66
5.2 RQ2 – Accuracy of Classic Profile Projection Heuristics 67
5.3 RQ3 – Comparison with LLM-Based Profile Prediction 68
5.4 RQ4 – Comparison with LLM-Based Profile Projection 70
5.5 RQ5 – Impact of Different Optimizations on Profile Projection 71
5.6 RQ6 – Running Time of the Different Heuristics 73
5.7 RQ7 – Running Time per Benchmark of the Classic Projection Heuristics . 74

6 Related Work **76**
6.1 Binary Diffing 76
6.2 Inter-Procedural Correspondence 77
6.3 Intra-Procedural Correspondence 77
6.4 Static Profile Prediction with Neural Networks 78

7 Conclusion **80**
7.1 Future Work 80

Bibliography **82**

Chapter 1

Introduction

Profile-guided optimization (PGO) is a widely used technique to improve program performance. It has been implemented in production compilers such as GCC [21], LLVM/Clang [40], and Microsoft Visual C++ [51], and is routinely applied in large software systems [48, 46]. By leveraging dynamic execution profiles collected from representative workloads, PGO enables optimizations such as inlining, loop unrolling, and code layout decisions to be tailored to the actual behavior of programs [26]. This approach can yield substantial performance improvements, with speedups of up to 20–30% being reported in both academic studies and industrial practice [44, 45, 38, 39].

PGO introduces a fundamental tension between optimization and information fidelity: profile data guides transformations that alter the very control-flow graph (CFG) to which the data refers. This leads to a chicken-and-egg problem. On the one hand, profile data is indispensable for effective optimization; on the other, the compiler optimizations often invalidates or distorts the profile. Once the CFG is transformed, the original mapping between execution counts or branch probabilities and program edges may no longer hold, leaving subsequent passes with stale or misleading information.

Modern compilers attempt to address this challenge, but the solutions often require nontrivial engineering effort. In LLVM, for example, many optimization passes contain custom logic to update or repair profile information as transformations are applied. Concrete instances include the function `updatePredecessorProfileMetadata` in `JumpThreading.cpp`[1], which refines predecessor probabilities during control-flow duplication. In other cases, when transformations are too disruptive, profile information is simply invalidated, as observed in LLVM's `simplifycfg` pass, for instance. Consequently, the need to preserve profile fidelity across a wide variety of optimizations has been the subject of repeated discussions on the LLVM mailing lists, often resulting in large and intricate patches that require careful design and extensive review.

There are techniques that attempt to project profile data from one version of a program onto another [2, 39, 54]. In principle, these methods could be used to propagate profile information along the optimization pipeline. However, the three techniques we are

[1] Available in `https://github.com/llvm/llvm-project/blob/a77c4948a5681984accd3c6d35fb51c1c5571a50/llvm/lib/Transforms/Scalar/JumpThreading.cpp#L148` on December 16th, 2025.

aware of were not designed with compiler optimizations in mind. Their primary goal is to enable the reuse of profile data during the program development cycle.

The approaches of Wang et al. [54] (BMAT) and Ayupov et al. [2] (SPM) project a profile from an older version of a program P onto a newer version P' by matching hash codes of basic blocks. This matching is exact and therefore fragile: even small transformations that alter the instructions inside basic blocks break the correspondence. To mitigate this problem, BMAT [54] and SPM [2] use a hierarchy of matches. They first hash all instructions of each basic block, and if no match is found, they fall back to a relaxed hash that considers only the opcodes of instructions. Along a similar direction, Moreira et al. [39] (Beetle) use, as hash codes, *branch features*, such as the opcode of a comparison (less-than, greater-than, etc.), the direction of the edge (backward or forward), and similar attributes[2]. Nevertheless, as we will show in Chapter 5, exact matching, even with relaxed hash functions, leads to an excessive number of misses in the presence of compiler optimizations.

1.1 The Contribution of This Work

This work proposes solutions to propagate profile data through compiler optimizations. Specifically, we address the following problem: given a program P, a sequence S of its basic blocks sorted by observed execution frequency, and an optimized version P' of P, how can we construct a sequence S' for P', also sorted by execution frequency, without executing P'? This work investigates two classes of heuristics to derive S':

- **Static Prediction:** $P' \mapsto S'$. These techniques analyze P' to infer the execution order S' using stochastic or predictive methods, without looking neither into P nor into S:
  - **Random:** randomly shuffle the blocks of P' to produce S' (see Section 3.1).
  - **LLVM:** apply the static profiling heuristics available in LLVM to infer S' (see Section 3.2).
  - **LLM-Pred:** use a large language model (LLM) to predict S' (see Section 3.3).
- **Static Projection:** $(P, S, P') \mapsto S'$. These techniques attempt to match the edges of the control-flow graphs (CFGs) of P and P', and use this mapping, together with S, to reconstruct S':

[2]Beetle's [39] approach was designed to preserve profile during development cycles, not compiler optimizations. In their own words: "*we expect (but do not require) that the set of modified vertices be much smaller than the total number of vertices.*"

- **Hash:** apply the hierarchy of hash functions proposed from SPM [2] to match CFG edges (see Section 4.2).
- **Histogram:** compare histograms of basic block instructions using Euclidean distance as a similarity criterion to match CFG edges (see Section 4.3).
- **LLM-Proj:** use an LLM to infer S', given knowledge of P, P', and S (see Section 4.5).

Some of these heuristics rely on matching criteria proposed in prior work, such as those of SPM [2] (currently in use in the BOLT binary optimizer [45]). However, to the best of our knowledge, they have not been previously evaluated as mechanisms for propagating profile information through compiler optimizations—a gap that this work addresses for the first time. Furthermore, the heuristics in Sections 3.3, 4.3 and 4.5 are novel.

We have implemented the heuristics proposed in this work within the LLVM compilation infrastructure (release 18.1.8). Chapter 5 demonstrates that histograms of LLVM instructions, despite their simplicity, provide highly accurate matching: they outperform all the other heuristics in almost every setting, including the hierarchy of hash functions proposed in prior work. This approach even surpasses the resource-intensive LLM-based technique, which leverages GPT-4.o to reconstruct stale profile data. Section 5.6 further shows that the runtime overhead of applying these heuristics is small enough to make them practical during compilation, while compiler optimizations are applied. In summary, this work makes the following observation:

Key Observation: The Euclidean distance between histograms of instructions in basic blocks is an effective criterion for matching blocks across versions of the same control-flow graph transformed by different optimizations. This heuristic is simpler to implement than previous approaches and is more resilient to program transformations, because it works by *similarity*, instead of *exact matching*.

To support the key observation, this work brought forward the following contributions:

- **The Matching Problem:** To ease the task of comparing different profile prediction or profile project approaches, Section 2.3 introduces a simple profiling problem: predicting the "hot-order" of basic blocks in a program's control-flow graph. This problem simplifies the comparison of different heuristics, as it removes measurement fluctuations from accuracy reports.
- **Similarity Search:** Section 4.3 proposes a new technique for matching basic blocks between two versions of a program, built upon two main ideas. First, it uses the Euclidean distance between opcode histograms (computed over a code region and

its neighborhood) as the matching criterion. Second, it traverses these regions recursively, matching them according to their dominance level. For example, the least nested regions are matched first, followed by their subregions, and so on.

- **LLM-based profiling:** The paper shows how a state-of-the-art LLM can be used either to predict a profile from scratch or to project an existing profile onto a new version of the program. Adapting an LLM to this task was non-trivial, since the textual representation of a program's control-flow graph often exceeds the capacity of the model's context window.

1.1.1 Software and Publications

The product of this work is the implementation of heuristics and scripts for projecting profile information. The code base is publicly available under the GPL 3.0 license and can be retrieved at: https://github.com/lac-dcc/hydra. Additionally, this work inspired the paper entitled "Automatic Propagation of Profile Information through the Optimization Pipeline", accepted for presentation on the annual ACM conference on Object-Oriented Programming, System, Languages & Applications [19].

Chapter 2

Preliminary Definitions

This chapter presents the fundamental concepts to understand the topics covered in this dissertation, starting from the notion of control-flow graphs (Section 2.1), going through the concepts of program profiles (Section 2.2), the hot-order problem (Section 2.3), static profile prediction (Section 2.4) and projection (Section 2.5).

2.1 Control-Flow Graph

During compilation, there are many ways in which a compiler represents a program. These representations are called Intermediate Representation (IR), and some common representations are abstract syntax tree and three address code. One form of IR is the Control-Flow Graph (CFG), which is defined in Definition 2.1.

Definition 2.1 (Control-Flow Graph). *A* control-flow graph (CFG) *of a program is a directed graph $G = (V, E)$ where:*

- *V is a set of vertices, each representing a* basic block*, which are a sequence of instructions without branches.*
- *E is a set of directed edges (u, v), where an edge indicates that block v may execute immediately after block u.*

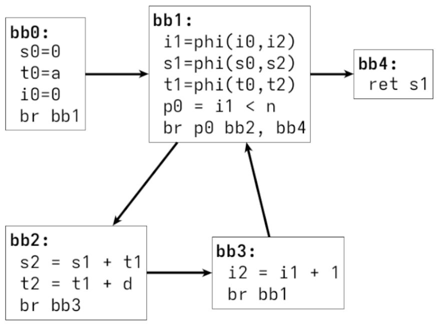
The CFG enables compilers to identify optimization opportunities. Even greater optimizations are possible when the CFG is in *Static Single-Assignment (SSA) form* [11], defined in Definition 2.2. Example 2.1 illustrates both a CFG and its SSA form, while Example 2.2 demonstrates how optimizations transform CFGs. These examples will be referenced throughout this work.

Definition 2.2 (Static Single-Assignment Form). *A program is in* Static Single-Assignment (SSA) form *if every variable has only one definition point. This property enables more ag-*

gressive optimizations. When multiple definitions may reach a point depending on control flow, a ϕ-function merges them into a single definition.

Example 2.1. *Figure 2.1(a) shows a function that computes the sum of an arithmetic progression iteratively. Part (b) presents the control-flow graph of the program*[1].

(a)
```
long arith_prog_sum(long a, long d, long n){
  long s = 0;
  long t = a;
  for (long i = 0; i < n; i++) {
    s += t;
    t += d;
  }
  return s;
}
```

(b)

```
bb0:
  s0=0
  t0=a
  i0=0
  br bb1

bb1:
  i1=phi(i0,i2)
  s1=phi(s0,s2)
  t1=phi(t0,t2)
  p0 = i1 < n
  br p0 bb2, bb4

bb2:
  s2 = s1 + t1
  t2 = t1 + d
  br bb3

bb3:
  i2 = i1 + 1
  br bb1

bb4:
  ret s1
```

Figure 2.1: (a) Function computing the sum of an arithmetic progression via iterative summation. (b) The respective function's CFG in SSA form.

Example 2.2. *Figure 2.2 shows optimizations applied to the CFG from Figure 2.1(b). Part (a) presents the Global Value Numbering (GVN) optimization, which reduces redundancy and thus the CFG size. Part (b) shows loop rotation, which increases the CFG size to enable further optimizations. Both are standard LLVM optimizations.*

2.2 Program Profile

The CFG of a program can be equipped with profile information of a program, which is acquired through its execution. The profile information can be used to apply more aggressive optimizations. We formalize the notion of a program profile in Definition 2.3 and illustrate it in Example 2.3.

Definition 2.3 (Profile Information). *An* edge profile *is a function $P_e : E \mapsto \mathbb{N}$ that maps each edge in E to its observed execution frequency. A* block profile *is a function $P_b : V \mapsto \mathbb{N}$ that maps each vertex in V to its observed execution frequency. The law of conservation of flow applies: for any $v \in V$, the sum of the frequencies of incoming edges*

[1]We use a simplified version of the LLVM IR. It preserves the Single-Static Assignment form [11] but omits types and auxiliary metadata.

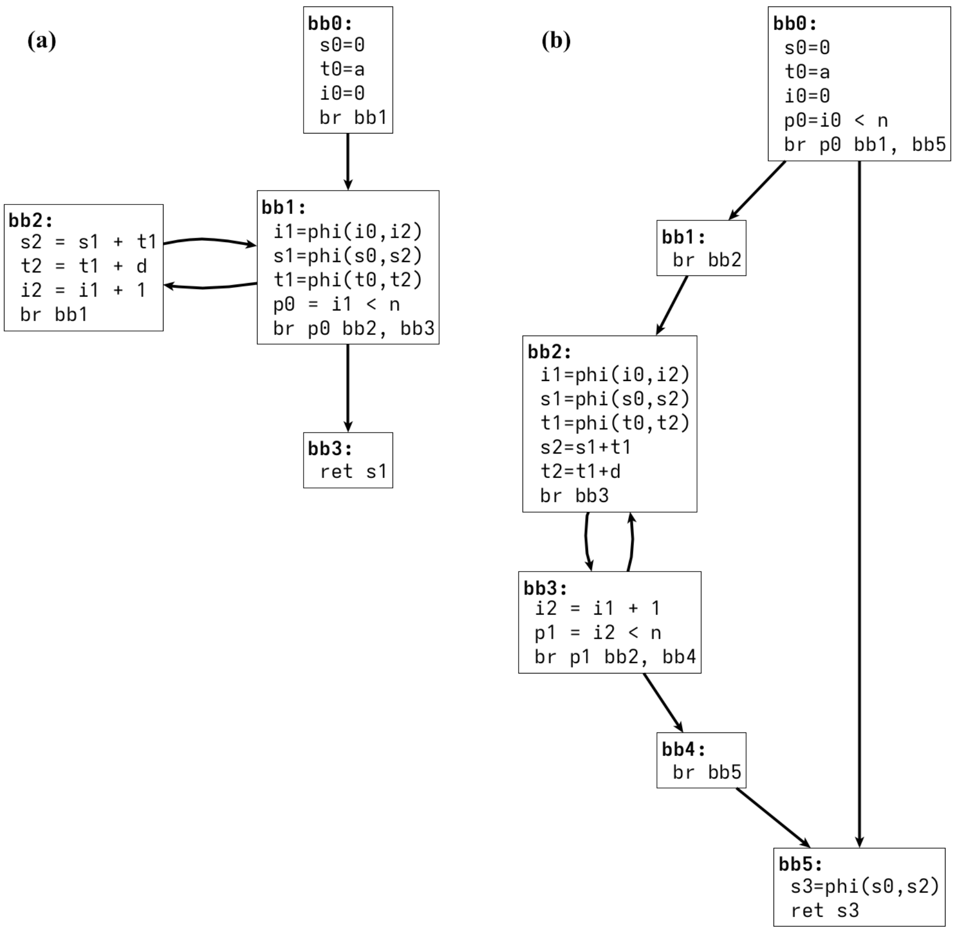


Figure 2.2: Optimized versions of the function from Figure 2.1: (a) after GVN optimization, (b) after loop rotation optimization.

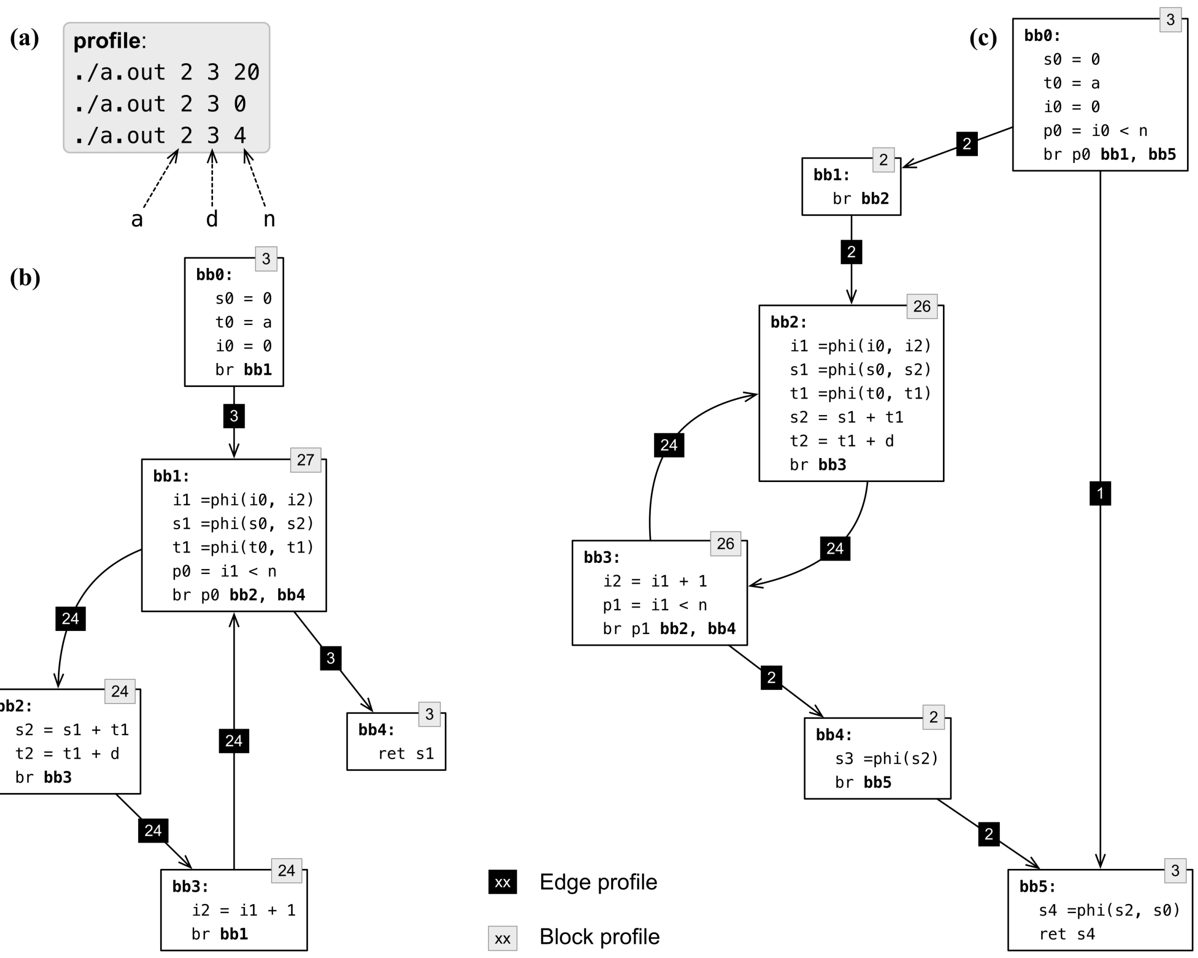
must equal the block frequency, which in turn equals the sum of the frequencies of outgoing edges, except at entry and exit nodes:

$$\sum P_e(u \to v) \;=\; P_b(v) \;=\; \sum P_e(v \to w). \tag{2.1}$$

Example 2.3. *Figure 2.3 (a) shows three different executions of a compiled version of the function from Figure 2.1 (a). Part (b) presents the control-flow graph annotated with profile information obtained from these executions. Part (c) shows the CFG of the same program after loop rotation with profile data. This data comes from the three executions in Figure 2.3 (a).*

Example 2.3 shows that a program profile, whether edge-based or block-based, is a total function: every block (or edge) of the program's CFG is assigned a frequency. Total profiles can be produced by two common methods, which will be detailed in the following subsections:

- **Instrumentation:** Counters are added to basic blocks, in order to measure how


```
profile:
./a.out 2 3 20
./a.out 2 3 0
./a.out 2 3 4
```
```
bb0:
  s0 = 0
  t0 = a
  i0 = 0
  br bb1

bb1:
  i1 =phi(i0, i2)
  s1 =phi(s0, s2)
  t1 =phi(t0, t1)
  p0 = i1 < n
  br p0 bb2, bb4

bb2:
  s2 = s1 + t1
  t2 = t1 + d
  br bb3

bb3:
  i2 = i1 + 1
  br bb1

bb4:
  ret s1
```
```
bb0:
  s0 = 0
  t0 = a
  i0 = 0
  p0 = i0 < n
  br p0 bb1, bb5

bb1:
  br bb2

bb2:
  i1 =phi(i0, i2)
  s1 =phi(s0, s2)
  t1 =phi(t0, t1)
  s2 = s1 + t1
  t2 = t1 + d
  br bb3

bb3:
  i2 = i1 + 1
  p1 = i1 < n
  br p1 bb2, bb4

bb4:
  s3 =phi(s2)
  br bb5

bb5:
  s4 =phi(s2, s0)
  ret s4
```

Figure 2.3: (a) Three executions producing profile data. (b) Control-flow graph of the program with profile information. (c) Optimized CFG after loop rotation, with profile data carried forward.

many times each edge was traversed. This method provides precise data but suffers from high overhead in larger programs. [18]

- **Sampling:** The program state is captured periodically during execution. This reduces the overhead, but may lose precision. [23]

2.2.1 Instrumentation-Based Profiling

Instrumentation-based profiling counts the number of times each basic block or edge executes [5]. This approach produces an instrumented version of the code with counters inserted at CFG edges. However, the modified program can be significantly slower than the original, making this technique not fitting for industrial use.

Knuth and Stevenson [30] introduced an optimization to reduce overhead by observing that a full profile can be reconstructed by instrumenting only the edges in the complement of the CFG's minimum spanning tree (MST). This greatly reduces the number of counters needed, though the approach still incurs high overhead.

A more recent advancement by Frenot and Pereira [18] replaces counters within loops with program variables that can count how many times the loop executes. This approach not only reduces the number of counters but also minimizes the execution cost of frequently executed counters. While this represents significant progress in overhead reduction, instrumentation still cannot match the collection performance of sample-based profiling.

Despite the higher overhead, instrumentation provides exact profile data [18], making it suitable for generating the ground truth used in our experiments.

2.2.2 Sample-Based Profiling

Sample-based profiling addresses the need for lower-overhead profiling by leveraging Performance Monitoring Units (PMUs) available in modern processors [23]. However, PMUs can slow program execution, leading processor designers to implement trade-offs that result in imprecise sampling [8, 56, 58, 23].

To mitigate these inaccuracies, recent research has employed advanced processor features such as Intel's *Precise Event Based Sampling* [8] and *Last Branch Records* technology [42, 9, 45]. Despite these efforts, inherent design limitations in modern processors prevent complete accuracy [56, 58], necessitating post-processing to correct sampled profiles. [23]

The first post-processing technique to reduce profile inaccuracy was introduced by Levin et al. [31], who proposed an approach based on the Minimum-Cost Flow (MCF) problem. However, this method propagated all possible flow along the same path, ignoring the original profile distribution. Subsequent works adapted MCF to propagate flow while maintaining the inferred profile similar to the original [9, 41, 43]. He et al. [23] added a final adjustment step to propagate profiles to blocks and edges not covered in the first MCF phase.

As noted in the SPM [2] paper, these correction techniques are useful not only for improving the precision of sampled profiles but also for correcting profiles propagated across program versions, as discussed in Section 4.4.

2.3 The Hot-Order Problem

We now introduce a benchmark for measuring the quality of a statically reconstructed profile, based on the fact that many optimizations use the hotness of blocks and paths [32]. This benchmark, called the *Hot-Order Problem*, has the advantage of being well defined both for static projection and for static prediction of profile information. Thus, it provides a unified way to evaluate all the heuristics described in Sections 3 and 4. The Hot-Order Problem is defined as follows:

Definition 2.4 (The Hot-Order Problem)**.** *The problem of ordering the blocks in a CFG by their* temperature *is defined as:*

- ***Input:*** *A control-flow graph $G = (V, E)$ and a block profile $P_b : V \mapsto \mathbb{N}$.*
- ***Output:*** *An ordering $L = [v_1, v_2, \dots, v_n]$ of V, where v_i denotes the i-th block in the sequence. Given two blocks v_i and v_j in L, we say that v_i is hotter than v_j if $i < j$.*
- ***Goal:*** *Let $S_R = [P_b(v_1), \dots, P_b(v_n)]$ be a reference ordering of V determined by P_b, i.e., $P_b(v_i) \geq P_b(v_j)$ for $i < j$. Let S_L be the sequence of execution frequencies of blocks in L, e.g., $S_L = [P_b(L_1), \dots P_b(L_n)]$. Let D be the* Minimum Swap Distance[2] *between S_L and S_R. The objective of the Hot-Order Problem is to minimize D.*

Ordering blocks by their temperature serves as a benchmark for the accuracy of profile reconstruction techniques. In other words, Definition 2.4 can be used to evaluate: (i) methods that reconstruct execution frequencies of basic blocks [39]; (ii) approaches that complete missing information in sample-based profilers [2, 23]; or (iii) techniques that estimate branch probabilities [7, 38]. Example 2.4 illustrates how this definition can be applied in practice.

Example 2.4 (Block Ordering by Hotness)**.** *Using the block frequencies P_b from Example 2.3, a reference hot-ordering of the program in Figure 2.3(c) is R = [`bb1`, `bb2`, `bb3`, `bb0`, `bb4`], with frequencies $S_R = [27, 24, 24, 3, 3]$. Consider instead a static inference approach that computes the ordering via a depth-first walk of the control-flow graph. This yields L = [`bb0`, `bb1`, `bb2`, `bb3`, `bb4`], with frequencies $S_L = [3, 27, 24, 24, 3]$. The Minimum Swap Distance in this case is 3, since S_L can be transformed into S_R using three swaps, e.g., (`bb0`, `bb1`), (`bb0`, `bb2`), (`bb0`, `bb3`).*

[2]The Minimum Swap Distance is the minimum number of swaps between adjacent elements required to transform L into R. For example, if $R = [a, b, c]$ and $L = [a, c, b]$, then the Minimum Swap Distance is 1 (one swap between c and b).

2.4 Static Profile Prediction

In the context of this work, the static profile prediction refers to techniques that attempt to infer the *hot-order* of a program's basic blocks by analyzing the program alone, without executing it. Definition 2.5 formalizes this problem.

Definition 2.5 (Static Profile Prediction)**.** *The problem of static profile prediction is defined as:*

- ***Input:*** *A control-flow graph $G = (V, E)$.*
- ***Output:*** *An ordering L of V, where $v = L_i \in V$ denotes the i-th block in the sequence.*
- ***Goal:*** *Minimize the minimum swap distance to a reference ordering L_R of G.*

A variety of techniques can be used to construct such an ordering, as explored in Section 3. In particular, any method that infers branch frequencies [55] can be used to reconstruct L. Alternatively, approaches that estimate branch probabilities [7, 38] are also applicable, since probabilities can be converted into execution frequencies [4].

2.5 Static Profile Projection

In the context of this work, the static profile projection refers to techniques that attempt to infer the hot-order of a program's basic blocks by mapping a profile produced for a reference control-flow graph onto another optimized version of the program, without executing this new version. We formalize this problem as follows:

Definition 2.6 (Static Profile Projection)**.** *The problem of static profile projection is defined as:*

- ***Input:*** *A control-flow graph $G_{ref} = (V_r, E_r)$, an edge profile $P_e : E_r \mapsto \mathbb{N}$, and a control-flow graph $G_{opt} = (V_{opt}, E_{opt})$ obtained after a finite sequence of transformations applied to G_{ref}.*
- ***Output:*** *An ordering L of V_{opt}, where $v = L_i \in V_{opt}$ denotes the i-th block in the sequence.*
- ***Goal:*** *Minimize the minimum swap distance to a reference ordering L_R of G_{opt}.*

Heuristics for profile projection are discussed in Section 4. Several of these heuristics form the core contribution of this paper, as they provide compiler engineers with practical means to propagate profile data across the optimization pipeline.

Chapter 3

Profile Prediction Heuristics

This chapter focuses on the different heuristics that we have designed and implemented to solve the Profile Prediction Problem of Definition 2.5. Thus, these heuristics produce an ordering of the basic blocks of a program, given only a static view of that program's control-flow graph, without any prior knowledge about it.

3.1 Random Order

This heuristic produces a random ordering of the basic blocks using the Mersenne Twister algorithm [33] as the random number generator. Example 3.1 shows how this simple heuristic works.

Example 3.1. *Suppose we have a program with the following basic blocks:* $[bb_0, bb_1, bb_2, bb_5]$. *This heuristic have an equal chance of returning any permutation of these elements, like* $[bb_0, bb_2, bb_1, bb_5]$, $[bb_5, bb_1, bb_2, bb_1]$ *or even* $[bb_0, bb_1, bb_2, bb_5]$.

Although this approach has no practical industrial value, it serves as a *null hypothesis*. A natural null hypothesis for an ordering heuristic is that it performs no better than an uniformly random ordering of the blocks. Theorem 3.1 gives the expected minimum swap distance under that null model: $\mathbb{E}[\text{minimum-swap-distance}] = N(N-1)/4$. If a heuristic yields an average minimum swap distance that is lower than this baseline (and the difference is statistically significant under an appropriate test), then one can reject the null hypothesis and conclude that the heuristic captures nontrivial structure of the program beyond random chance.

Theorem 3.1. *Let* $N \geq 1$ *and consider the set of all permutations of* N *distinct elements. If a permutation is sampled uniformly at random, the expected minimum swap distance (i.e., the expected minimum number of adjacent swaps required to transform the* *permutation into the sorted order, equivalently the expected number of inversions) equals*

$$\mathbb{E}[\textit{minimum-swap-distance}] \;=\; \frac{N(N-1)}{4}.$$

Proof. Index the N distinct elements by $1, \dots, N$ in the sorted order. For each pair $1 \le i < j \le N$ let the indicator random variable

$$X_{ij} = \begin{cases} 1 & \text{if the pair } (i,j) \text{ is inverted in the sampled permutation (i.e., } j \text{ appears before } i\text{),} \\ 0 & \text{otherwise.} \end{cases}$$

The total number of inversions (equivalently the minimum swap distance) is

$$X \;=\; \sum_{1 \le i < j \le N} X_{ij}.$$

By linearity of expectation,

$$\mathbb{E}[X] \;=\; \sum_{1 \le i < j \le N} \mathbb{E}[X_{ij}].$$

For any fixed pair (i,j), in a uniformly random permutation the two relative orders $i \prec j$ and $j \prec i$ are equally likely, hence $\mathbb{P}(X_{ij} = 1) = \frac{1}{2}$. Therefore $\mathbb{E}[X_{ij}] = \frac{1}{2}$, and

$$\mathbb{E}[X] \;=\; \binom{N}{2} \times \frac{1}{2} \;=\; \frac{N(N-1)}{2} \times \frac{1}{2} \;=\; \frac{N(N-1)}{4},$$

which proves the theorem. □

3.2 LLVM Predictor

This heuristic leverages the LLVM infrastructure to estimate the execution frequency of basic blocks. It relies on the `BranchProbabilityInfoAnalysis` pass, which computes branch probability estimates for conditional branches and switch statements in a function. The analysis annotates CFG edges with probabilities indicating how likely each successor of a terminator instruction is to be taken. When available, these probabilities are derived from profile-guided optimization (PGO) data. In the absence of PGO (which is the case relevant to our work) they are obtained from static heuristics. These heuristics are based on features similar to those described by Ball and Larus [4], such as: loop backedges are likely to be taken, error-handling paths are unlikely, and equality tests (`==`) tend to evaluate to false.

Given these statically assigned probabilities, we compute a hot-order of basic blocks in three steps:

1. **Initialization:** Compute the probability of each edge being taken using the `BranchProbabilityInfoAnalysis` pass.

2. **Propagation:** Propagate execution frequencies throughout the CFG using the Wu-Larus algorithm [55]. This algorithm propagates frequencies from the innermost loops to the outermost loops. We extend it by adding a propagation phase aiming at the complete CFG.

3. **Ordering:** Sort the basic blocks by their estimated frequencies.

Algorithms 1 and 2 present the pseudocode for the propagation algorithm. The former details the propagation subroutine, while the latter executes the calls to propagate the estimated frequencies.

Algorithm 1 executes only if the current block, bb, has not been visited, and all of its predecessors, $pred$, have either been visited or form a back edge ($pred \to bb$). This algorithm first iterates through the predecessors of bb to compute the estimated frequency $P_b[bb]$. It then processes the successors of bb to update the estimated frequencies of its outgoing edges and the probability of taking a back edge.

To compute $P_b[bb]$, the algorithm uses the concept of cyclic probability, defined as the probability of a loop being taken. This probability relies on the probability of a back edge being taken, which is explained later. The cyclic probability is calculated as follows:

$$cyclic\_probability = \sum_{\substack{p \in pred(bb) \\ (p \to bb) \text{ is a back edge}}} back\_edge\_probability[e]$$

The cyclic probability can exceed 100% in cases of loops that may not terminate. For such cases, the algorithm defines an upper bound for the cyclic probability of $1 - \epsilon$. While Wu and Larus [55] does not specify a value for ϵ, we found that $\epsilon = 0.03$ yields the best results in our tests.

From the predecessors of bb that do not form a back edge to bb, we compute an auxiliary block frequency:

$$P'_b[bb] = \sum_{\substack{p \in pred(bb) \\ (p \to bb) \text{ is not a back edge}}} P_e[(pred \to bb)]$$

Using this auxiliary frequency, the algorithm calculates $P_b[bb]$ as:

$$P_b[bb] = \begin{cases} 1 & \text{if } bb = head \\ \frac{P'_b[bb]}{1 - cyclic\_probability} & \text{otherwise} \end{cases}$$

Subsequently, the algorithm marks bb as visited and estimates the edge frequency P_e for each successor $succ$. This is calculated as the traversal probability of the edge

$(bb \to succ)$ multiplied by $P_b[bb]$. If the edge $(bb \to succ)$ is a back edge (i.e., $succ$ is the loop header), then the $back\_edge\_prob[(bb \to succ)]$ is assigned the value of $P_e[(bb \to succ)]$.

Finally, for each successor $succ$ of bb where the edge $(bb \to succ)$ is not a back edge, the algorithm recursively calls the function `propagate(`$succ, head$`)`.

In Algorithm 2, the calls to the propagation function are made first from the innermost to the outermost loop headers. Subsequently, a final call is made with the function's entry block. This final call, which is our extension to the original Wu-Larus algorithm, ensures the propagation of edge frequencies to blocks in acyclic regions of the CFG. Example 3.2 shows how the algorithm works.

Example 3.2. *Figure 3.1 illustrates how the LLVM predictor works on the program from Example 2.3(a).*

Part (a) shows the initial state of the program, where all frequencies and edge probabilities are unknown.

Part (b) presents the branch probabilities computed using ***`BranchProbability-InfoAnalysis`****. These probabilities are derived from LLVM's static heuristics. For instance, loop backedges are assumed to be very likely, which explains the probability of 0.96875 assigned to the edge* $(bb_3 \to bb_1)$*, while the loop exit edge* $(bb_3 \to bb_4)$ *receives a much lower probability of 0.03125.*

Part (c) shows the result of applying the Algorithm 1 to the loop composed of basic blocks $[bb_1, bb_2, bb_3]$*. At this stage, the estimated frequencies are real values in the interval* $[0, 1]$*. This happens because the algorithm will only assign a probability to the backedge* $(bb_3 \to bb_1)$ *after estimating the frequency of* bb_1*. Without this probability, it cannot estimate a frequency correspondent to the loop.*

Part (d) presents the final frequencies after applying the Algorithm 1 to the entire program. Now, with the backedge probability of 0.96875 established in the previous part, the algorithm can compute the loop's frequency. The frequency of the loop header bb_1 *is calculated as* $\frac{1}{1-0.96875} = \frac{1}{0.03125} = 32$*. This frequency is then propagated to* bb_1*'s successors:* $32 \times 0.96875 = 31$ *is propagated to* bb_2*, and* $32 \times 0.03125 = 1$ *is propagated to* bb_4*. The remaining frequencies are propagated along edges* $e = (src \to dst)$ *with* 100% *of probability of being taken, meaning the frequency is assigned as* $P_e[e] = P_b[src]$*.*

Algorithm 1: Wu-Larus propagation algorithm, propagation subroutine.

```
Input: A function CFG $G$, a branch probability mapping from an edge to a
       double value $Prob : e \to p$
Output: Predicted profile for blocks $P_b$ and edges $P_e$ on $G$

Function propagate(bb, head):
    $P_b, P_e \leftarrow \emptyset$
    if bb has not been visited then
        if $bb = head$ then
            $P_b[bb] \leftarrow 1$
        else
            foreach predecessor pred of bb do
                if pred has not been visited and edge ($pred \to bb$) is not a back edge
                 then
                    return
                end
            end
            $P_b[bb] \leftarrow 0$
            $cyclic\_prob = 0$
            foreach predecessor pred of bb do
                if edge ($pred \to bb$) is a back edge then
                    $cyclic\_prob = cyclic\_prob + back\_edge\_prob[(pred \to bb)]$
                else
                    $P_b[bb] \leftarrow P_b[bb] + P_e[(pred \to bb)]$
                end
            end
            if $cyclic\_prob > 1 - \epsilon$ then
                $cyclic\_prob = 1 - \epsilon$
            end
            $P_b[bb] \leftarrow \frac{P_b[bb]}{1 - cyclic\_prob}$
        end
        mark bb as visited
        foreach successor succ of bb do
            $P_e[(bb \to succ)] \leftarrow Prob[(bb \to succ)] \times P_b[bb]$
            if $succ == head$ then
                $back\_edge\_prob[(bb \to succ)] \leftarrow P_e[(bb \to succ)]$
            end
        end
        foreach successor succ of bb do
            if edge ($bb \to succ$) is not a back edge then
                $P'_b, P'_e \leftarrow$ propagate($succ, head$)
                $P_b \leftarrow P_b \cup P'_b$; $P_e \leftarrow P_e \cup P'_e$
            end
        end
    end
    return $P_b, P_e$
```

Algorithm 2: Wu-Larus propagation algorithm.

```
Input: A function CFG $G$, a branch probability mapping from an edge to a
       double value $Prob : e \to p$
Output: Predicted profile for blocks $P_b$ and edges $P_e$ on $G$

Initialize $P_b, P_e \leftarrow \emptyset$
foreach edge e do
    $back\_edge\_prob[e] \leftarrow Prob[e]$
end
foreach loop l from inner-most to out-most do
    $head \leftarrow header(l)$
    mark all blocks reachable from head as not visited
    mark all other blocks as visited
    $P'_b, P'_e \leftarrow$ propagate($head, head$)
    $P_b \leftarrow P_b \cup P'_b$; $P_e \leftarrow P_e \cup P'_e$
end
$entry \leftarrow entry\_block(G)$
mark all blocks reachable from entry as not visited
mark all other blocks as visited
$P'_b, P'_e \leftarrow$ propagate($entry, entry$)
return $P_b \cup P'_b, P_e \cup P'_e$
```

3.3 Prediction with a Generative Model

This heuristic uses a large generative model to predict the hot-ordering of basic blocks. We implemented the analysis using Microsoft GenAIScript, a TypeScript-based framework for composing prompt-driven AI workflows. The configuration was as follows:

- **Model:** OpenAI GPT-4o (Vision variant, text-only mode)
- **Deployment:** Azure OpenAI Service
- **Identifier:** `azure:gpt-4o_2024-11-20`
- **Parameters:** Max tokens: 16,384; temperature: 0 (deterministic); other hyperparameters not publicly disclosed
- **Response format:** JSON with a strict schema
- **Documentation:** `https://ai.azure.com/catalog/models/gpt-4o`

To automate model interaction, we developed a GenAIScript workflow that processes LLVM IR programs (`.ll` files). The workflow executes the following steps:

1. Filter the input files and extract functions using regular expressions on `define` blocks.

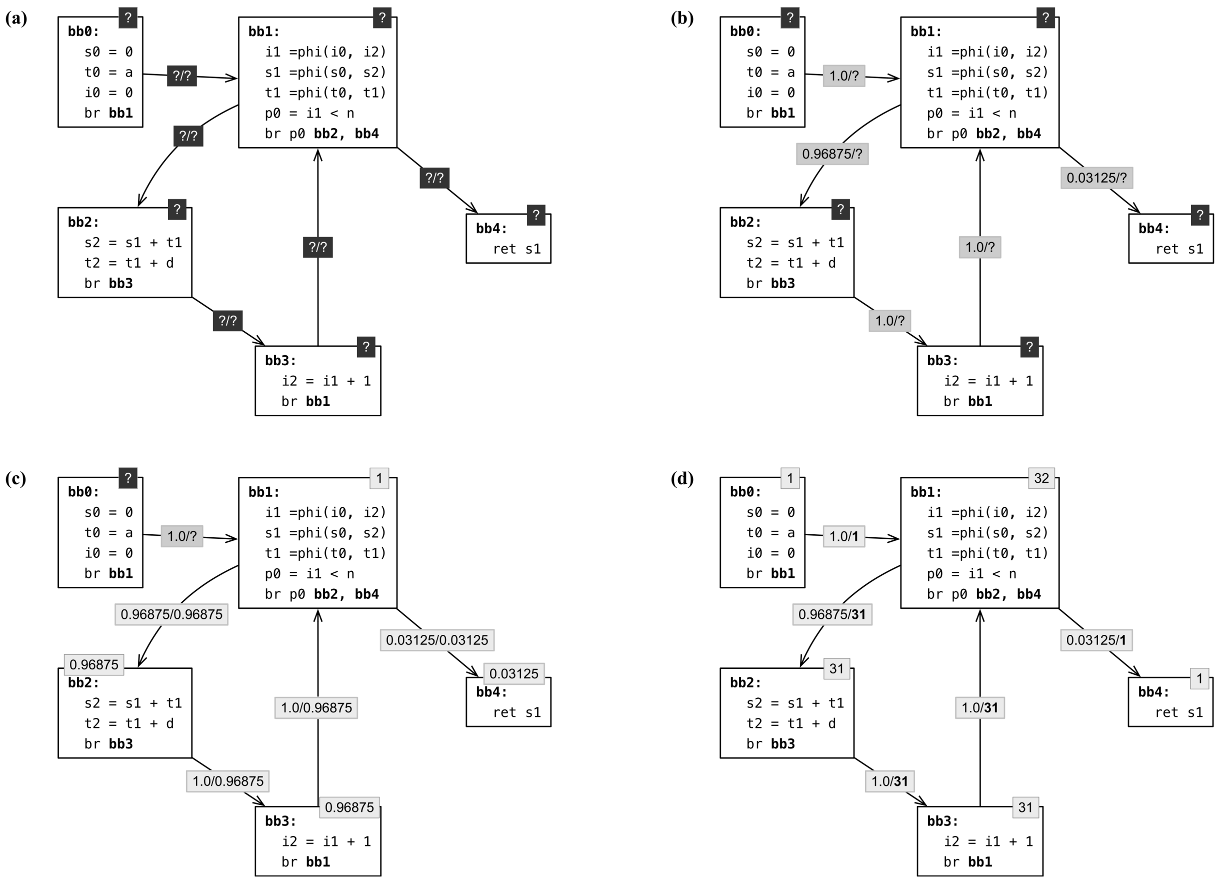


Figure 3.1: Static inference of profile information using the branch probability heuristics available in the LLVM compiler. Numbers associated with each block show the frequency of execution of that block. The pairs *prob/freq* associated with each edge show the probability that the edge is traversed and its execution frequency, i.e., how often the edge is expected to be traversed.

2. Parse the labels of basic blocks in each function.

3. Generate a tailored prompt for each function following the template in Figure 3.2.

4. Send the prompt to GPT-4o via GenAIScript's `runPrompt` interface, enforcing a JSON schema with three fields:

   - `bbOrderByHotness`: ordered list of blocks, with explanations.
   - `benchmarkInfo`: identifiers for the file/function.
   - `additionalNotes`: optional observations.

5. Parse the model's response and save it to structured JSON files, one per benchmark.

```
Role: You are a Compiler Engineer with over 20 years of experience  in compiler construction,
LLVM internals, and advanced optimizations techniques. You have deep expertise in low-level
code generation, IR transformations, and performance tuning for modern architectures, with a
strong track record in static analysis, JIT compilation, and custom backend development.

Task: You are given a function written in LLVM’s intermediate representation. You are asked
to analyze the function and provide insights on its basic blocks.

I have two questions regarding this function **${completeFunc}**:

Question 1, Hot Block Estimation:

Could you guess which basic block would be the "hottest",  where "hottest" means the basic
block that is more likely to be executed more often?

Question 2, Hotness Ranking:

This function has ${bbSet.length} basic blocks: ${bbSet.join(", ")}. Could you sort this
sequence of basic blocks by "hotness"? That is, if a basic block bb_x appears ahead of
another block bb_y, it means you think bb_y will not be executed more often than bb_x, though
they might have equal frequency.

Reasoning and Heuristics: Please provide a brief explanation of your reasoning for the
hottest basic block and the order of the other basic blocks as you did. If relevant, include
any insights from control-flow structure, loop presence, loop header, branching behavior, or
any other heuristics you applied during your analysis.
```

Figure 3.2: Prompt template used to request a profile prediction from GPT-4o. The template asks the model to identify the hottest basic block and to produce a ranking of blocks by hotness, along with reasoning.

Chapter 4

Profile Projection Heuristics

This chapter presents heuristics for solving the Profile Projection Problem (Definition 2.6). In contrast to the prediction heuristics of Section 3, projection techniques operate with richer information. In addition to the target control-flow graph G_{opt}, they also take as input a previous version of the program, G_{ref}, along with a corresponding edge profile P_e. Note that G_{ref} and G_{opt} may differ in both the number of vertices and overall structure.

4.1 A General Projection Template

With the exception of the LLM-based heuristic discussed in Section 4.5, all heuristics in this chapter follow the same template, which Algorithm 3 shows.

The algorithm takes the CFGs G_{ref} and G_{opt}, and the profile P_e as input. It begins by initializing two empty data structures: a block matching M_b and an auxiliary edge profile P'_e for G_{opt}.

The block matching M_b is defined by a function `block_matching`. The definition of this function is how the heuristics explored in Sections 4.2 and 4.3 are differentiated. Section 4.2 describes a hash-based method for matching based on the SPM [2] paper, while Section 4.3 presents a histogram-based method, which is an original contribution of this work.

Using the matching M_b, the algorithm then proceeds to match edges between G_{ref} and G_{opt}. An edge $e_{ref} \in G_{ref}$ is matched to an edge $e_{opt} \in G_{opt}$ if the source block of e_{ref} is matched to the source block of e_{opt} and the target block of e_{ref} is matched to the target block of e_{opt}. For each matched edges, the algorithm assigns $P'_e[e_{opt}] = P_e[e_{ref}]$.

Finally, for any remaining unmatched edges, the algorithm infers their profile by running a Minimum Cost Flow algorithm via the `fill_missing_data`, following the ideas from He et al. [23] (Profi), producing a complete profile P''. This process is explained in detail in Section 4.4.

Algorithm 3: Profile projection via block and edge matching.

```
Input: Original program $G_{ref}$ with profile $P_e$, optimized program $G_{opt}$
Output: Projected profile $P''$ on $G_{opt}$

Initialize $M_b \leftarrow \emptyset$, $P'_e \leftarrow \emptyset$
$M_b \leftarrow$ block_matching($G_{ref}, G_{opt}, \ldots$ additional parameters $\ldots$)
foreach edge $e_{ref} \in G_{ref}$ do
    if $\exists e_{opt} \in G_{opt}$ such that $\mathrm{src}(e_{opt}) = M_b[\mathrm{src}(e_{ref})]$ and $\mathrm{dst}(e_{opt}) = M_b[\mathrm{dst}(e_{ref})]$
     then
        $P'_e[e_{opt}] \leftarrow P_e[e_{ref}]$
    end
end
Use Profi [23] to extend $P'_e$ to a total function: $P'' \leftarrow$ fill_missing_data($P'_e$)
```

4.2 Hash-Based Matching

Two prior techniques, BMAT [54] and SPM [2], address the Profile Projection Problem by matching hashes derived from the syntax of basic blocks. These methods rely on a hierarchy of hash functions to balance collision resistance and robustness to program transformations. In this work, we focus on the technique from SPM [2], which uses the following three hash variants to find a match for a basic block *bb*:

Loose: a hash based solely on the ordered set of instruction opcodes in *bb*, ignoring operands.

Strict: a hash based on all instruction opcodes and operands in *bb*, in their exact order of appearance.

Full: a combination of *bb*'s strict hash and the loose hashes of its predecessors and successors.

To match basic blocks between two versions of a control-flow graph (e.g., G_{ref} and G_{opt} in Definition 2.6), SPM [2] first finds candidate blocks via loose matching. It then prioritizes matches based on the full hash, followed by the strict hash, using the block's position as a final tiebreaker. Algorithm 4 implements this pattern.

The algorithm starts by initializing the block matching M_b as an empty set. It then iterates through each basic block $bb_{opt} \in G_{opt}$, creating a mapping from opcode hashes to the list of blocks that share that hash. The opcode hash of a block is computed from the string formed by concatenating the elements of the ordered set of opcodes of its instructions. For instance, a block with three `sub` instructions and one `br` instruction would have its hash computed over the string `"brsub"`.

Subsequently, the algorithm iterates through each basic block $bb_{ref} \in G_{ref}$, trying to match them with a basic block in the optimized program. First, it retrieves the candidate list $c \subseteq G_{opt}$ that have the same opcode hash as bb_{ref}, excluding the basic blocks that have already been matched. If c is empty and bb_{ref} is the entry block of G_{ref}, it's matched with the entry block of G_{opt}; otherwise it is marked as unmatched.

Finally, if c is not empty, the algorithm matches bb_{ref} with the unmatched block $bb_{opt} \in c$ that has the smallest distance to it. The distance metric implements the hash hierarchy: a difference in predecessor or successor opcode hashes (affecting the full hash) incurs a large penalty of 2^{32}, while a difference in the strict instruction hash incurs a smaller penalty of 2^{16}. This ensures that a full hash match is prioritized over a strict hash match. As a tiebreaker for blocks with equal distances, the algorithm uses the relative position of bb_{ref} and bb_{opt} within their respective CFGs. Example 4.1 illustrates how this approach works in practice.

Example 4.1. *Figure 4.1 (a) shows the hash codes that SPM [2] assign to the program from Figure 2.3 (a). For clarity, only the last four digits of each decimal hash value are displayed. Part (b) shows a new version of the program, optimized with LLVM's implementation of Global Value Numbering (GVN) [47]. In LLVM 18.1.8, this optimization merges two blocks from Figure 4.1 (a), e.g., `bb2` and `bb3`, into a single block `bb2` in Figure 4.1 (b). The full hash successfully matches three blocks between the two versions: `bb0`, `bb1`, and `bb4`. The merged block `bb2` in Fig. 4.1 (b) remains unmatched under both the full and strict hashes. However, the loose hash is able to establish a correspondence between `bb2` in Figure 4.1 (a) and `bb2` in Figure 4.1 (b).*

4.3 Histogram-Based Matching

This section introduces a heuristic that matches basic blocks based on *structural similarity*. This approach contrasts with the exact hash matching of BMAT [54] and SPM [2], described in Section 4.2.

Our similarity criterion is based on the Euclidean distance between *opcode histograms*. This section is divided into two parts: first, we give the definitions that are used in this heuristic; then, we explain the algorithms used to run the heuristic.

Algorithm 4: Hash-based block matching.

```
Input: Original program $G_{ref}$, optimized program $G_{opt}$
Output: Basic block matching $M_b$

Function distance($bb_{ref}, bb_{opt}$):
    $neigh_{ref} \leftarrow$ hash_succ($bb_{ref}$) $+$ hash_pred($bb_{ref}$)
    $instr_{ref} \leftarrow$ hash_instr($bb_{ref}$)
    $neigh_{opt} \leftarrow$ hash_succ($bb_{opt}$) $+$ hash_pred($bb_{opt}$)
    $instr_{opt} \leftarrow$ hash_instr($bb_{opt}$)
    $order\_dist =$ abs(order($bb_{ref}$) - order($bb_{opt}$))
    return
     $2^{32} \times (neigh_{ref} == neigh_{opt}) + 2^{16} \times (instr_{ref} == instr_{opt}) + order\_dist$

// Main process begins here
Initialize $M_b \leftarrow \emptyset, Op\_Blocks \leftarrow \emptyset$
foreach basic block $bb_{opt} \in G_{opt}$ do
    $op\_hash =$ hash_opcode($bb_{opt}$)
    $Op\_Blocks[op\_hash] \leftarrow Op\_Blocks[op\_hash] \cup \{bb_{opt}\}$
end
foreach basic block $bb_{ref} \in G_{ref}$ do
    $G'_{opt} \leftarrow \{bb'_{opt} \in G_{opt} \mid \exists bb'_{ref} \in G_{ref} \mid M_b[bb'_{ref}] = bb'_{opt}\}$
    $c = Op\_Blocks[$hash_opcode($bb_{ref}$)$] \setminus G'_{opt}$
    if $c \neq \emptyset$ then
        $best\_distance = \infty$
        foreach basic block $bb_{opt} \in c$ do
            $dist =$ distance($bb_{ref}, bb_{opt}$)
            if $dist < best\_distance$ then
                $M_b[bb_{ref}] \leftarrow bb_{opt}$
                $best\_distance = dist$
            end
        end
    else
        if $bb_{ref} = entry\_block(G_{ref})$ then
            $M_b[bb_{ref}] \leftarrow$ entry_block($G_{opt}$)
        else
            $M_b[bb_{ref}] \leftarrow$ undef
        end
    end
end
```

4.3.1 Opcode Histogram Definitions

This section defines the notion of opcode histograms, and how they are constructed from a basic block and from a loop[1] $L = \{bb_1, bb_2, \ldots, bb_k\}$. Then follow the definition of

[1]A loop is a strongly connected component in a control-flow graph with a single entry point.

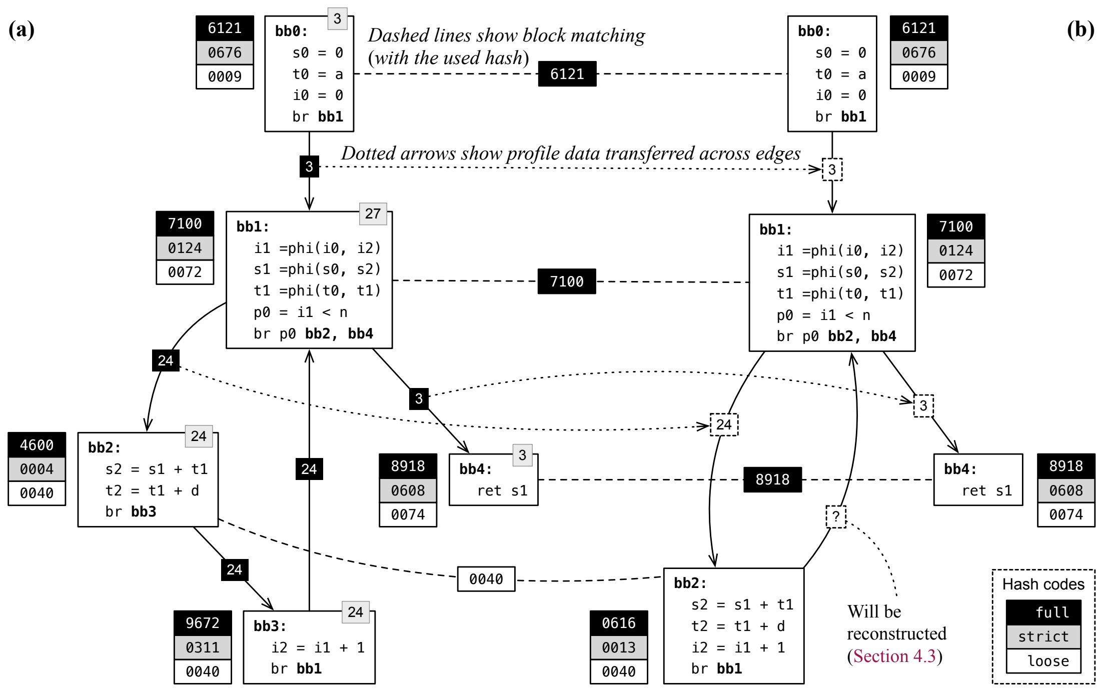

(a)
(b)
Dashed lines show block matching (with the used hash)
Dotted arrows show profile data transferred across edges
bb0: s0 = 0 t0 = a i0 = 0 br bb1
bb1: i1 =phi(i0, i2) s1 =phi(s0, s2) t1 =phi(t0, t1) p0 = i1 < n br p0 bb2, bb4
bb2: s2 = s1 + t1 t2 = t1 + d br bb3
bb3: i2 = i1 + 1 br bb1
bb4: ret s1
bb2: s2 = s1 + t1 t2 = t1 + d i2 = i1 + 1 br bb1
6121 0676 0009
7100 0124 0072
4600 0004 0040
9672 0311 0040
8918 0608 0074
0616 0013 0040
Will be reconstructed (Section 4.3)
Hash codes
full
strict
loose

Figure 4.1: (a) The program from Figure 2.3 (a), annotated with three hash variants for each basic block. (b) The same program after optimization with LLVM's Global Value Numbering.

the sum operation over two opcode histograms.

Definition 4.1 (Opcode Histograms). *Let I be a set of instructions from an opcode alphabet L_{op}. An* Opcode Histogram, *denoted H_I, is a vector of size $|L_{op}|$, where each element $H_I[o]$ counts the number of times that opcode $o \in L_{op}$ occurs in I.*

Definition 4.2 (Block and Loop Histograms). *The* Block Histogram *H_{bb} for a basic block bb is the opcode histogram constructed over its instruction set, I_{bb}.*

The Loop Histogram *H_L for a loop is the opcode histogram of the union of all instructions within its basic blocks, $I_L = \bigcup_{i=1}^{k} bb_i$.*

Note: *To simplify Algorithm 6, we treat a single basic block outside any loop as a loop of size one, providing a non-recursive base case.*

Definition 4.3 (Sum of Opcode Histograms). *Let H_a and H_b be two opcode histograms of size k. Their sum, denoted $H_a \oplus H_b$, is a new histogram of size k where each element is the sum of the corresponding frequencies: $\forall 1 \le i \le k, (H_a \oplus H_b)[i] = H_a[i] + H_b[i]$.*

To add more context from the control-flow graph to an opcode histogram, and enable a more precise matching, we define notions of predecessors and successors histogram for an opcode histogram.

Definition 4.4 (Predecessor and Successor Histogram)**.** *The* Predecessor Histogram *of a block (or loop) with histogram H is the sum of the histograms of all its predecessors: $pred(H) = \bigoplus_{p \in preds} H_p$.*

The Successor Histogram*, $succ(H)$, is defined analogously as the sum of its successors' histograms.*

Note: *For a loop L, its predecessors and successors are defined over the control-flow graph with its strongly connected components contracted.*

Finally, we can define how to compute the distance between two opcode histograms, which is the metric used for comparing them by similarity.

Definition 4.5 (Opcode Histogram Distance)**.** *Let H_a and H_b be two opcode histograms. The distance between them, denoted $d(H_a, H_b)$, is the sum of the Euclidean distances between the histograms themselves and their neighbors:*

$$d(H_a, H_b) = E_d(H_a, H_b) + E_d(pred(H_a), pred(H_b)) + E_d(succ(H_a), succ(H_b))$$

where the euclidean distance $E_d(A, B)$ between two vectors A and B of size k is defined as:

$$E_d(A, B) = \sqrt{\sum_{i=1}^{k} (A_i - B_i)^2}$$

4.3.2 Matching Blocks by Similarity

Histogram-based matching establishes correspondences between the basic blocks of G_{ref} and G_{opt} through a two-stage process, using two algorithms:

- `loop_matching` (Section 4.3.2.2): Matches loops between the two CFGs using loop histograms as the similarity criterion.
- `block_matching` (Section 4.3.2.3): Matches basic blocks between the two CFGs using block histograms as the similarity criterion.

In the template of Algorithm 3, the `block_matching` function is replaced with the following function call:

$$M_b \leftarrow \texttt{block\_matching}(G_{ref}, G_{opt}, \texttt{loop\_matching}(G_{ref}, G_{opt}, null, null))$$

This two-stage approach ensures that larger program structures (loops) are matched first, providing context that guides and constrains the subsequent matching of basic blocks. Next, we dive into the details of each algorithm, first explaining an auxiliary function used in both algorithms, and then discussing the `loop_matching` and `block_matching` functions.

4.3.2.1 The Closest Histogram Function

A fundamental operation used by both `loop_matching` and `block_matching` is finding the most similar histogram from a candidate set. This operation is implemented by the function `closest_histogram`, described in Algorithm 5.

The function takes a set of target opcode histograms H and a source histogram h_s. Notice that these opcode histograms can refer to either block or loop histograms. Its goal is to find the histogram $h_t \in H$ that minimizes the distance $d(h_s, h_t)$, where d is the opcode histogram distance from Definition 4.5. The subroutines `distance` and `euclidean` in the algorithm implement the formulae that constitute this metric.

4.3.2.2 The Loop Matching Function

Some optimizations may change the structure of loops within the CFG, which is the case of `loop-rotate`. From previous empirical evaluation, only the block matching (Section 4.3.2.3) is not sufficient to yield good results over such optimizations. Hence, we propose this function to incremented this phase to the histogram-based matching to deal with such optimizations.

To match loops between the programs G_{ref} and G_{opt}, we implement a recursive strategy described in Algorithm 6. Example 4.2 illustrates how the algorithm works. The algorithm starts by matching outer loops, and then proceeds to match inner loops, ensuring that structural context is established at each level. The algorithm takes four parameters:

- The reference and optimized program, G_{ref} and G_{opt}.
- A loop $L_{ref} \in G_{ref}$ and a loop $L_{opt} \in G_{opt}$ representing the current loops being matched.

The loop parameters L_{ref} and L_{opt} can be `null`, which indicates that the algorithm should match the top-level loops (or basic blocks outside of loops treated as ”loops” of

Algorithm 5: Closest Histogram

```
Input: Set of opcode histograms $H$, source histogram $h_s$
Output: Opcode histogram $h_t \in H$ that minimizes $d(h_s, h_t)$

Function euclidean($A$, $B$):
    if $|A| \neq |B|$ then
        return $\infty$
    end
    $sum \leftarrow 0$
    for $i \leftarrow 1$ to $|A|$ do
        $sum \leftarrow sum + (A[i] - B[i])^2$
    end
    return $\sqrt{sum}$

Function distance($h_a$, $h_b$):
    $hist\_d \leftarrow$ euclidean$(h_a, h_b)$
    $pred\_d \leftarrow$ euclidean$(pred(h_a), pred(h_b))$
    $succ\_d \leftarrow$ euclidean$(succ(h_a), succ(h_b))$
    return $hist\_d + pred\_d + succ\_d$

// Main process begins here
$h_t \leftarrow$ null
$min\_dist \leftarrow \infty$
foreach $h' \in H$ do
    $current\_dist \leftarrow$ distance$(h_s, h')$
    if $current\_dist < min\_dist$ then
        $h_t \leftarrow h'$
        $min\_dist \leftarrow current\_dist$
    end
end
return $h_t$
```

size one) at the program level.

The output is a mapping M_b between the basic blocks of L_{ref} and L_{opt}. In the initial call where both loops are `null`, the output is a mapping between all blocks in G_{ref} and G_{opt} that belong to matched loops or acyclic regions.

The algorithm begins by initializing two empty mappings: M_b for blocks and M_l for loops. It then collects the sets of nested loops, L'_{ref} and L'_{opt}, contained within L_{ref} and L_{opt}, respectively. If the input loops are `null` (indicating the execution of loop matching over the full program), L'_{ref} and L'_{opt} are initialized to the set of top-level loops in their respective CFG. As previously defined, this set includes any loops and treats single basic blocks outside any loop as loops of size one.

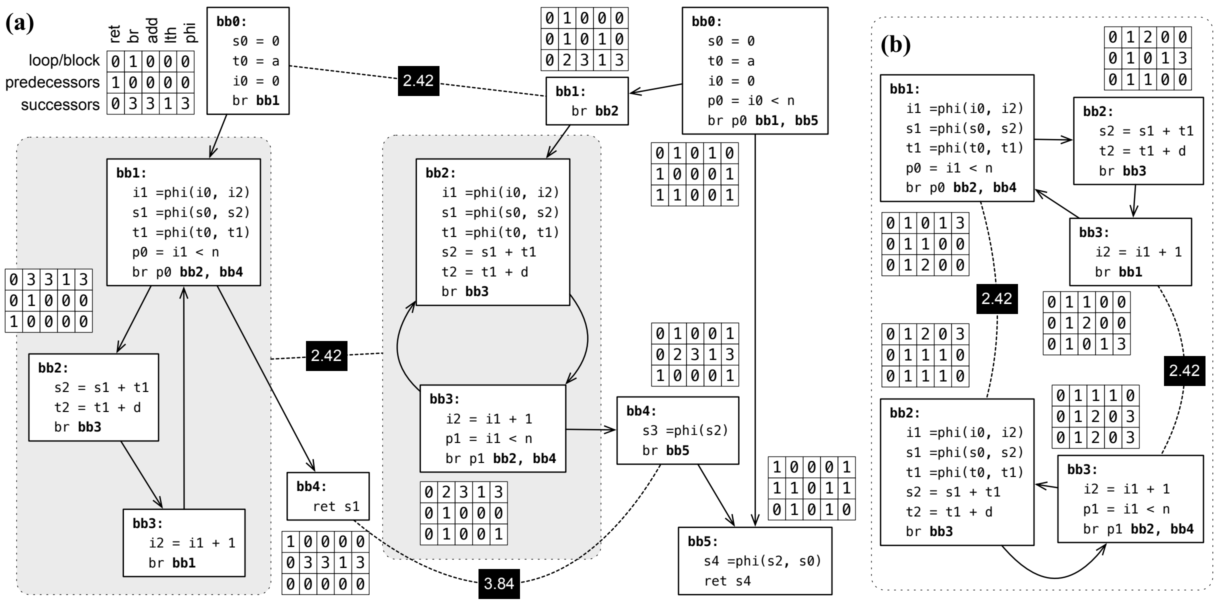
Then, it iterates through each loop $l_{ref} \in L'_{ref}$ and, using Algorithm 5, finds the closest unmatched loop $l_{opt} \in L'_{opt}$. The matched loops are mapped in the matching M_l, in a similar process to the block matching phase of Algorithm 4.

Finally, the algorithm iterates through each matched pair $(l_{ref}, l_{opt}) \in M_l$, doing one of the following operations:

- If $|l_{ref}| > 1$ (i.e., it is indeed a loop), the algorithm recursively calls `loop_matching` $(G_{ref}, G_{opt}, l_{ref}, l_{opt})$ to match the loops and blocks nested inside them. The resulting block mappings are merged into M_b.
- If $|l_{ref}| = 1$ (i.e., it is a single basic block), the algorithm directly maps the header block of l_{ref} to the header block of l_{opt} in M_b.

Algorithm 6: Histogram-Based Loop Matching.

```
Input: Reference program $G_{ref}$, optimized program $G_{opt}$, reference loop $L_{ref}$,
       optimized loop $L_{opt}$
Output: Matching of blocks $M_b$ from $L_{ref}$ to $L_{opt}$
Function loop_matching($G_{ref}, G_{opt}, l_{ref}, l_{opt}$):
    Initialize $M_b \leftarrow \emptyset$; $M_l \leftarrow \emptyset$
    if $L_{ref} = null$ then
        Initialize $L'_{ref}$ as the top-level loops of $G_{ref}$
        Initialize $L'_{opt}$ as the top-level loops of $G_{opt}$
    else
        Initialize $L'_{ref}$ as the loops inside $L_{ref}$
        Initialize $L'_{opt}$ as the loops inside $L_{opt}$
    end
    foreach loop $l_{ref} \in L'_{ref}$ do
        $L'' \leftarrow \{M_l[l'_{ref}] \mid l'_{ref} \in L'_{ref}$ and $M_l[l'_{ref}]$ is defined$\}$
        $l_{opt} \leftarrow$ closest_loop($L'_{opt} \setminus L'', l_{ref}$)
        if $l_{opt} \neq \emptyset$ then
            $M_l[l_{ref}] \leftarrow l_{opt}$
        else
            if $l_{ref} = entry\_loop(L_{ref})$ then
                $M_l[l_{ref}] \leftarrow$ entry_loop($L_{opt}$)
            else
                $M_l[l_{ref}] \leftarrow$ undef
            end
        end
    end
    foreach loop $l_{ref} \in L'_{ref}$ do
        let $l_{opt} \leftarrow M_l[l_{ref}]$
        if $l_{opt} \neq undef$ then
            if $l_{ref}$.size $= 1$ then
                $M_b[l_{ref}$.header$] \leftarrow l_{opt}$.header
            else
                $M_b = M_b \cup$ loop_matching($G_{ref}, G_{opt}, l_{ref}, l_{opt}$)
            end
        end
    end
```



```
bb0:
  s0 = 0
  t0 = a
  i0 = 0
  br bb1

bb1:
  i1 =phi(i0, i2)
  s1 =phi(s0, s2)
  t1 =phi(t0, t1)
  p0 = i1 < n
  br p0 bb2, bb4

bb2:
  s2 = s1 + t1
  t2 = t1 + d
  br bb3

bb3:
  i2 = i1 + 1
  br bb1

bb4:
  ret s1

bb0:
  s0 = 0
  t0 = a
  i0 = 0
  p0 = i0 < n
  br p0 bb1, bb5

bb1:
  br bb2

bb2:
  i1 =phi(i0, i2)
  s1 =phi(s0, s2)
  t1 =phi(t0, t1)
  s2 = s1 + t1
  t2 = t1 + d
  br bb3

bb3:
  i2 = i1 + 1
  p1 = i1 < n
  br p1 bb2, bb4

bb4:
  s3 =phi(s2)
  br bb5

bb5:
  s4 =phi(s2, s0)
  ret s4
```

Figure 4.2: (a) Matching of block regions using Algorithm 6 (`loop_matching`). We let G_{ref} be the left-side CFG, and G_{opt} the right-side one. (b) Matching of block regions after a recursive call of Algorithm 6.

Example 4.2. *Figure 4.2 shows the histogram matching process when projecting a profile from the program in Figure 2.3 (c) to its optimized version after loop rotation in Figure 2.3 (d).*

Figure 4.2 (a) shows the initial call to the `loop_matching` function with `null` loops. The histograms are simplified, displaying only the operands present in the basic blocks: `ret`, `br`, `add`, `lth`, and `phi`. In this phase, a matching is formed between the three regions `[bb0]`, `[bb1,bb2,bb3]` and `[bb4]` from the original program and the three regions `[bb1]`, `[bb2,bb3]` and `[bb4]` from the optimized program. The edges in the figure show the Euclidean Distance between matched histograms.

Subsequently, `loop_matching` is recursively called for the regions with multiple blocks: `[bb1, bb2, bb3]` (original) and `[bb2, bb3]` (optimized). This recursive step is shown in Figure 4.2(b). Here, the inner loops are matched, resulting in a mapping from `[bb1]` to `[bb2]` and from `[bb3]` to `[bb3]`.

4.3.2.3 The Block Matching Function

Some blocks may remain unmatched after running Algorithm 7, either because they belong to an unmatched loop or due to structural differences between the programs. Therefore, we implement Algorithm 7, which matches the basic blocks between G_{ref} and

G_{opt}, while preserving the matches that were established in the `loop_matching` function. The algorithm takes three parameters:

- The reference and optimized program, G_{ref} and G_{opt}.
- A preliminary block matching M'_B, generated from the application of `loop_matching`.

The output is a mapping M_b between the basic blocks of G_{ref} and G_{opt}. The algorithm's structure is similar to the first `foreach` block in Algorithm 6, but it operates on individual basic blocks instead of loops.

This algorithm may still leave some blocks unmatched (i.e., G_{ref} blocks associated with *undef* or G_{opt} blocks with no pre-image in G_{ref}). This occurs when G_{ref} and G_{opt} have a different number of basic blocks. In such cases, only the most similar blocks are matched. Example 4.3 illustrates this final matching step.

Algorithm 7: Histogram-Based Block Matching.

```
Input: Original program $G_{ref}$, optimized program $G_{opt}$, preliminary block
       matching $M'_b$
Output: Matching of basic blocks $M_b$ from $G_{ref}$ to $G_{opt}$

Initialize $M_b \leftarrow M'_b$
$B' \leftarrow \{b' \in G_{opt} \mid \exists b'' \in G_{ref} \mid M_b[b''] = b'\}$
foreach basic block $b_{ref} \in G_{ref} \mid b \notin M'_b \vee M'_b[b_{ref}] = undef$ do
    $b_{opt} \leftarrow$ closest_block($G_{opt} \setminus B', b_{ref}$)
    if $b_{opt} \neq \emptyset$ then
        $M_b[b_{ref}] \leftarrow b_{opt}$
        $B' \leftarrow B' \cup \{b_{opt}\}$
    else
        if $b_{ref} = entry\_block(G_{ref})$ then
            $M_b[b_{ref}] \leftarrow$ entry_block($G_{opt}$)
        else
            $M_b[b_{ref}] \leftarrow$ undef
        end
    end
end
```

Example 4.3. *After* `loop_matching` *executes on the program from Figure 4.2 (a), two blocks in the optimized program remain unmatched:* `bb0` *and* `bb5`*. Similarly, one block from the original program has no match:* `bb2`*. Algorithm 7 then pairs* `bb2` *from G_{ref} with* `bb5` *in G_{opt}, as this pair has the smallest Euclidean distance compared to other candidates. Note that matching a block inside a loop with a block outside any loop may introduce inaccuracies in the projected profile. However, such discrepancies are later corrected by the algorithm described in Section 4.4.*

4.4 Filling Out Missing Profile Information

After matching basic blocks and edges of G_{ref} and G_{opt}, Algorithm 3 assigns known profile values from P_e to the corresponding edges in G_{opt}, resulting in a partial profile P_e'. This section describes how to infer the remaining missing profile data to produce a complete and consistent profile P_e''.

We adopt the SPM [2], who build on top of Profi [23]. In their approach, the inference of missing profile data is reduced to an instance of the Minimum-Cost Flow problem [15]. The core idea is to find a valid flow (in respect to the law of flow conservation) that is as close to the known edge frequencies as possible. If a complete profile cannot be inferred, Profi [23] distribute the remaining flow according to a branch-probability heuristic. Like in the original SPM [2] algorithm, we use a uniform heuristic that assigns equal probability to all outgoing edges of a block.

Algorithm 8 shows an overview of the Profi [23] procedure. Examples 4.4 and 4.5 illustrate how this inference works. The subsequent subsections details each of its steps:

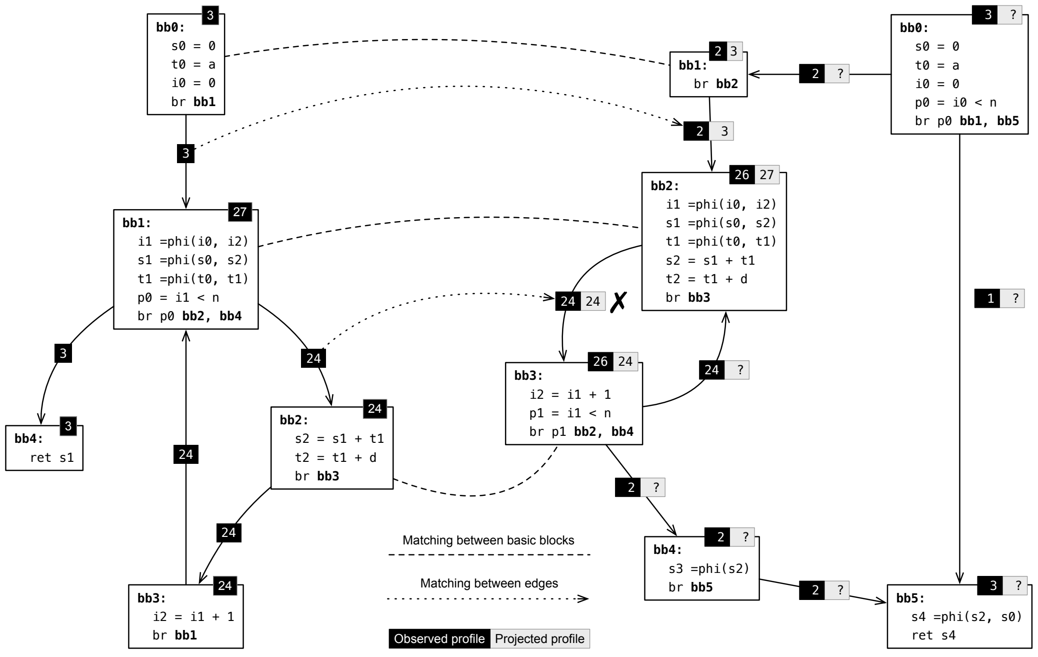
- **Auxiliary Network Construction** (Section 4.4.1): Building a flow network from the CFG.
- **Flow Augmentation** (Section 4.4.2): Applying a max-flow algorithm to satisfy flow conservation.
- **Extract Profile From Flow Network** (Section 4.4.3): Transform the flow network information into a edge profile.
- **Joining Isolated Components** (Section 4.4.4): Connecting disconnected parts of the flow network.
- **Rebalancing Flows** (Section 4.4.5): Final adjustments to handle residual inconsistencies.

Example 4.4. *Figure 4.3 shows how histogram-based block matching enables the transfer of profile information from a reference program to an optimized version. Here, blocks `bb0`, `bb1`, and `bb2` in G_{ref} are matched respectively with `bb1`, `bb2`, and `bb3` in G_{opt}. Profile data can be directly transferred between matched blocks, and between edges, when sources and destinations are matched. Blocks and edges without a match remain with undefined profile values (that must be reconstructed).*

Example 4.4 illustrates that direct projection of profile data can violate the law of conservation of flow. For instance, in Figure 4.3, the edge from `bb2` to `bb3` in G_{opt} is the

Algorithm 8: Filling missing profile information.

```
Input: Control-flow graph $G_{opt}$ with partial edge profile $P'_e$
Output: Completed profile $P''$ of $G_{opt}$
Use $P'_e$ and Equation 2.1 to build partial block profile $P'_b$
Initialize $(H, s, t) \leftarrow$ auxiliary_network($G_{opt}, P'_b, P'_e$)        // Section 4.4.1
while $X$ = find_augmenting_dag($H, s, t$) $\neq \emptyset$ do
    augment_flow_along_dag($H, s, t, X$)                                  // Section 4.4.2
end
$P''_e \leftarrow$ extract_weights($G_{opt}, H$)                                   // Section 4.4.3
$P''_e \leftarrow$ join_isolated_components($G_{opt}, P''_e$)                       // Section 4.4.4
$P'' \leftarrow$ rebalance_dangling_subgraphs($G_{opt}, P''_e$)                     // Section 4.4.5
return $P''$
```



bb0: s0 = 0 t0 = a i0 = 0 br bb1
bb1: i1 =phi(i0, i2) s1 =phi(s0, s2) t1 =phi(t0, t1) p0 = i1 < n br p0 bb2, bb4
bb2: s2 = s1 + t1 t2 = t1 + d br bb3
bb3: i2 = i1 + 1 br bb1
bb4: ret s1
bb0: s0 = 0 t0 = a i0 = 0 p0 = i0 < n br p0 bb1, bb5
bb1: br bb2
bb2: i1 =phi(i0, i2) s1 =phi(s0, s2) t1 =phi(t0, t1) s2 = s1 + t1 t2 = t1 + d br bb3
bb3: i2 = i1 + 1 p1 = i1 < n br p1 bb2, bb4
bb4: s3 =phi(s2) br bb5
bb5: s4 =phi(s2, s0) ret s4
Matching between basic blocks
Matching between edges
Observed profile
Projected profile

Figure 4.3: Matching between the control-flow graphs of Figure 2.3 (c–d), illustrating how profile data are transferred from the reference code G_{ref} (left) to the optimized code G_{opt} (right). Dashed arrows indicate inferred correspondences between basic blocks and edges. Question marks mark missing or undefined profile values.

only outgoing edge of `bb2`, yet its edge frequency does not match the block frequency; hence, breaking flow conservation. The reconstruction procedure of Profi [23] resolves such inconsistencies, as shown next.

Example 4.5. *Figure 4.4 shows how Profi [23] works on the left-side program seen in Figure 4.3. Some frequencies remain identical to the projected profile, while others, like those involving `bb0` and `bb5`, are adjusted to satisfy flow conservation. In particular, when a block or edge lacks a defined frequency, Profi [23] redistributes the missing flow proportionally according to branch probabilities.*

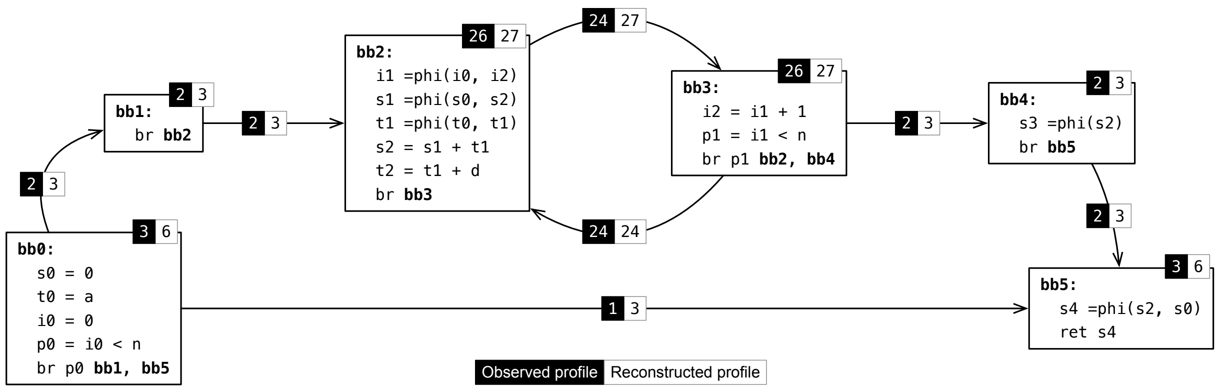


Figure 4.4: Result of applying Profi [23]'s reconstruction algorithm to the optimized program in Figure 4.3. The observed and reconstructed profiles are shown side by side, with adjusted frequencies ensuring flow conservation across the CFG.

4.4.1 Constructing an Auxiliary Network

The first step is to construct a flow network H from the control-flow graph G_{opt} and its partial profiles P'_b and P'_e. This network is designed to model profile inference as a Minimum-Cost Flow problem, where the goal is to find a flow that satisfies conservation constraints while minimizing the cost of deviations from the known profile values.

In this network, each edge is represented as a 4-tuple $(src \to dst, cap, cost)$. For simplicity, we omit the initial $flow$ value, as it is initialized to 0. The costs inc_b, dec_b, inc_j, and dec_j are parametrized in the general algorithm. Algorithm 9 details the construction process. The algorithm works as follows:

First, it initializes four special vertices: the main source s and sink t for the MCF problem, and auxiliary vertices s' and t' that connect to the CFG's entry and exit blocks.

The main idea of the construction is to split each basic block bb into an input node bb_I and an output node bb_O. This technique allows us to model flow conservation within the block and assign costs to the flow passing through it. By creating a primary edge $(bb_I \to bb_O)$ with infinite capacity and cost inc_b, we are allowing the algorithm to increase the block's execution count from its initial profile.

- If bb is an entry block, it connects the auxiliary source s' to bb_I.
- If bb is an exit block, it connects bb_O to the auxiliary sink t'.
- If bb has a known frequency $F = P'_b[bb]$, it adds three edges:
    1. A reverse edge $(bb_O \to bb_I)$ with capacity F and cost dec_b.
    2. An edge from the main source s to bb_O, with capacity F and no cost.
    3. An edge from bb_I to the main sink t, with capacity F and no cost.

Algorithm 9: Create an Auxiliary Flow Network

```
Input: Control-flow graph $G_{opt}$ with partial block profile $P'_b$ and partial edge
       profile $P'_e$
Output: Flow network $H$, with source $s$ and sink $t$

Initialize $s' \leftarrow 2 \times |G_{opt}|$
Initialize $t' \leftarrow s' + 1$
Initialize $s \leftarrow s' + 2$
Initialize $t \leftarrow s' + 3$
Initialize $H$ with $2 \times |G_{opt}| + 4$ vertices, source $s$ and sink $t$
foreach basic block $bb \in G_{opt}$ do
    $bb_I \leftarrow 2 \times bb$.index
    $bb_O \leftarrow bb_I + 1$
    $H$.addEdge($bb_I \rightarrow bb_O, \infty, inc_b$)
    if $bb$ is an entry block then
        $H$.addEdge($s' \rightarrow bb_I, \infty, 0$)
    end
    if $bb$ is an exit block then
        $H$.addEdge($bb_O \rightarrow t', \infty, 0$)
    end
    if $P'_b[bb] \neq undef$ then
        $H$.addEdge($bb_O \rightarrow bb_I, P'_b[bb], dec_b$)
        $H$.addEdge($s \rightarrow bb_O, P'_b[bb], 0$)
        $H$.addEdge($bb_I \rightarrow t, P'_b[bb], 0$)
    end
end
foreach edge $e \in G_{opt}$ do
    $H$.addEdge($src(e)_O \rightarrow dst(e)_I, \infty, inc_j$)
    if $P'_e[e] \neq undef$ then
        $H$.addEdge($dst(e)_I \rightarrow src(e)_O, P'_e[e], dec_j$)
        $H$.addEdge($s \rightarrow dst(e)_I, P'_e[e], 0$)
        $H$.addEdge($src(e)_O \rightarrow t, P'_e[e], 0$)
    end
end
$H$.addEdge($t' \rightarrow s', \infty, 0$)
return $(H, s, t)$
```

This setup effectively prescribes F units of flow through the block, but allows the MCF solver to adjust it, at a cost, if necessary to achieve global consistency.

For each CFG edge $e = (src \rightarrow dst)$, the algorithm adds a primary edge $(src_O \rightarrow dst_I, \infty, inc_j)$, representing the potential to increase the edge's profile. If the edge has a known frequency $F = P'_e[e]$, it also adds the analogous reverse edge and main source and sink connections.

Finally, an edge $(t' \rightarrow s', \infty, 0)$ is added. This completes the flow network into a circulation, ensuring that the flow entering through s' can return to s' via t', which is necessary for the MCF formulation. Example 4.6 illustrates this process.

Example 4.6. *This example shows how Algorithm 9 creates a flow network for the CFG and its matched profile from Figure 4.3 (right).*

Figure 4.5(a) shows the construction for basic blocks bb_0 and bb_5, which are entry and exit blocks. Since their profiles are undefined, as well as the edge ($bb_0 \rightarrow bb_5$), only the core block splitting edges, the primary edge between bb_{0_O} and bb_{5_I}, and the connections to the auxiliary source s' and sink t' are created.

Figure 4.5(b) shows the construction for basic blocks bb_1 and bb_2, which have defined profile through matching. Here, in addition to the core edges, the algorithm adds the reverse edges and main source and sink connections that encode the known frequencies.

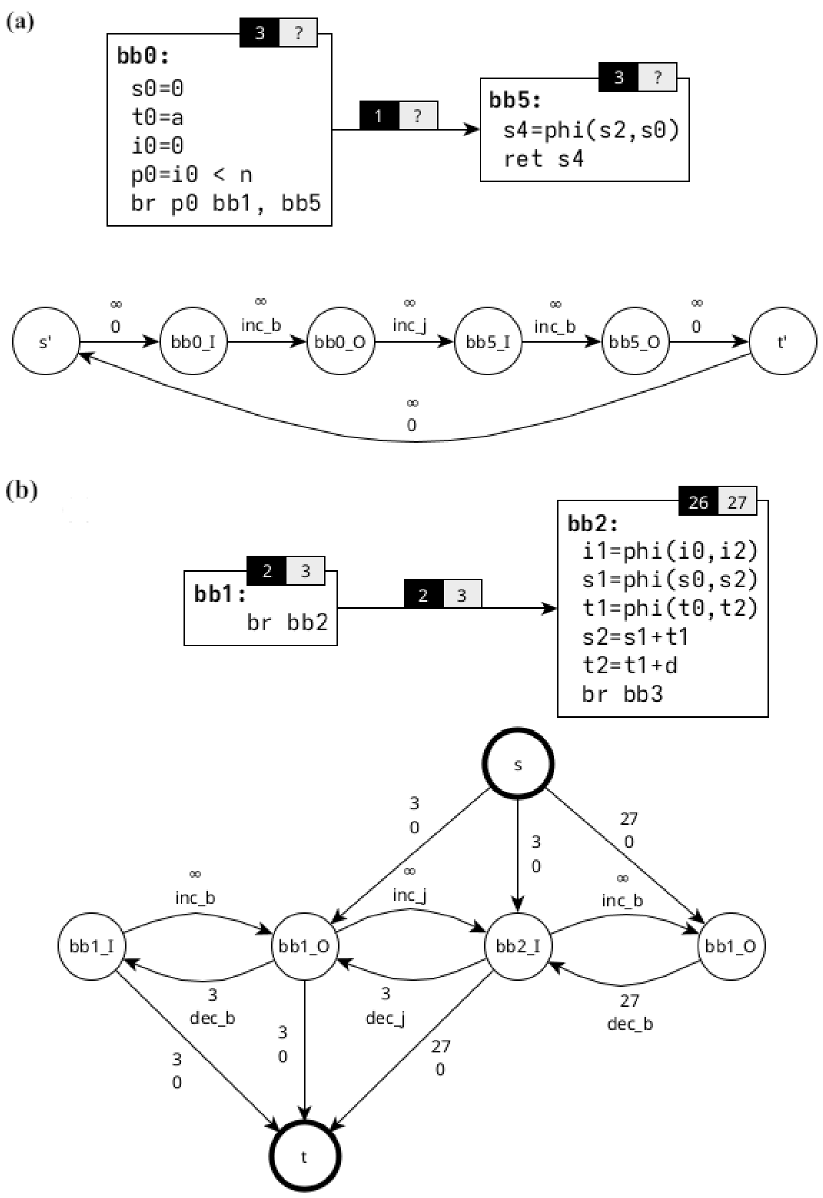
Figure 4.6 shows the complete flow network for the entire CFG. Edge capacities and costs are omitted for readability, but left to the curious reader as an exercise.

4.4.2 Running Flow Augmentation

After constructing the flow network H, the next step is to compute a flow that satisfies the flow conservation while minimizing cost. This is achieved through a minimum-cost flow algorithm, as outlined in the loop of Algorithm 8. The process iterates between finding an augmenting structure and pushing flow through it until no more flow can be sent from s to t. The maximum flow ensures flow conservation, while the minimum cost ensures the inferred profile remains as close as possible to the original.

The algorithm uses a modified Moore-Bellman-Ford approach [37] within a Ford-Fulkerson maximum flow algorithm [17], modified to use shortest paths instead of arbitrary paths, in order to minimize the cost. The process involves two main steps, repeated until no augmenting path exists:

- **Find an Augmenting DAG:** Find the shortest path from s to t and construct a Directed Acyclic Graph (DAG) containing all shortest paths of the same weight (Algorithm 10).
- **Augment Flow:** Push as much flow as possible through this DAG, distributing it as evenly as possible (Algorithms 11 and 12).



```
bb0:
  s0=0
  t0=a
  i0=0
  p0=i0 < n
  br p0 bb1, bb5
```

```
bb5:
  s4=phi(s2,s0)
  ret s4
```

```
bb1:
    br bb2
```

```
bb2:
  i1=phi(i0,i2)
  s1=phi(s0,s2)
  t1=phi(t0,t2)
  s2=s1+t1
  t2=t1+d
  br bb3
```

Figure 4.5: Excerpts from the flow network creation process for the CFG in Figure 4.3, showing the network for (a) basic blocks bb_0 and bb_5, and (b) basic blocks bb_1 and bb_2.

4.4.2.1 Finding an Augmenting DAG

Algorithm 10 finds candidate paths for augmentation. It implements a modified Moore-Bellman-Ford algorithm [37], which uses a worklist to relax vertices more efficiently than the original Bellman-Ford [6]. The algorithm initializes:

- $d[v]$: the shortest distance from source s to vertex v.
- $p[v]$: the predecessor of v on its shortest path from s.

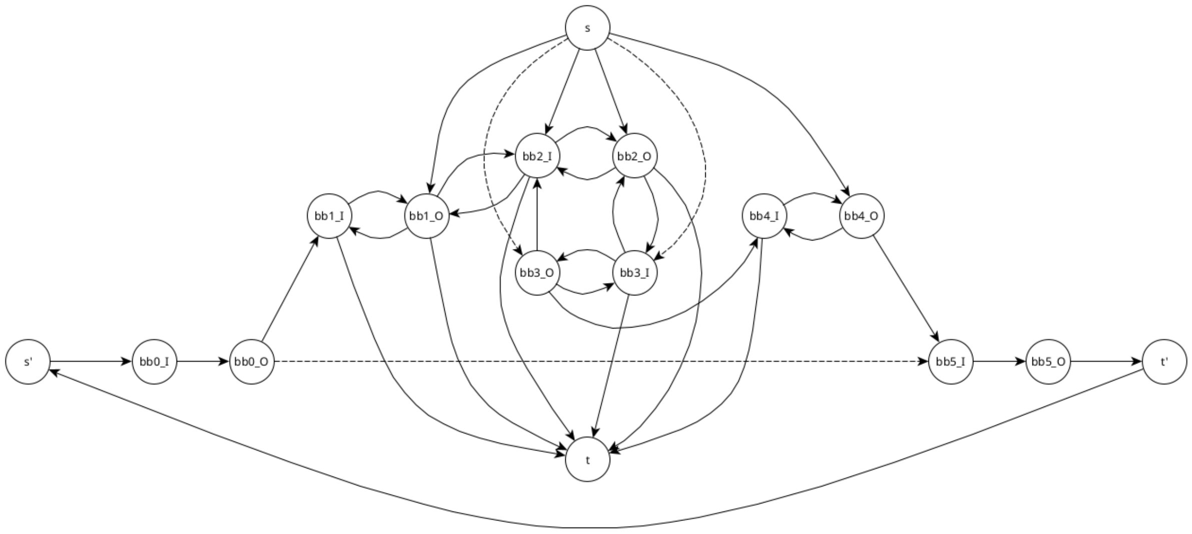


Figure 4.6: Complete flow network without edges capacities and costs for the CFG and matched profile from Figure 4.3 (right).

It then processes vertices from a queue, relaxing only edges that are not saturated ($flow < capacity$). This ensures we only consider edges that can carry more flow. Profi [23] guarantees that even with reverse edges in the residual graph, no negative cycles are formed.

Finally, the algorithm constructs a DAG $X \subseteq H$ containing all edges (u, v) that lie on *some* shortest path from s to t (i.e., where $d[v] = d[u] + cost$). This DAG represents all optimal paths for flow augmentation at the current cost.

4.4.2.2 Augmenting the Flow

Once we have an augmenting DAG X, Algorithm 11 runs the flow augmentation process. It first computes P_C, the maximum flow that can be sent along the shortest path from s to t (using the predecessor mapping p). Then, it repeatedly attempts to push flow through the entire DAG X using Algorithm 12. If this fails, it falls back to pushing the remaining flow along the single shortest path.

Algorithm 12 implements the augmentation along a DAG. It aims to distribute flow evenly across all paths of X, by running the following steps:

1. It computes F, the maximum flow that can be pushed through X, considering the proportion of flow that will be split through each edge.

2. It initializes ϕ, a mapping tracking flow for vertices and edges.

Algorithm 10: Find Augmenting DAG in Flow Network

```
Input: Flow network $H$ with source $s$ and sink $t$
Output: An augmenting DAG $X \subseteq H$
Initialize $d \leftarrow$ mapping from $v \in V(H)$ to shortest distance from $s$
Initialize $p \leftarrow$ mapping from $v \in V(H)$ to predecessor in shortest path
$d(s) \leftarrow 0$; $d(v) \leftarrow \infty, \forall v \in V(H) - \{s\}$
$p(v) \leftarrow \emptyset, \forall v \in V(H)$
Initialize $Q \leftarrow$ queue containing $s$
while $Q$ not empty and $d(t) > 0$ do
    $u \leftarrow Q$.pop()
    if $d(u) \leq d(t)$ then
        foreach edge $(u \rightarrow v, flow, cap, cost) \in E(H)$ do
            if $flow < cap$ and $d(u) + cost < d(v)$ then
                $d(v) \leftarrow d(u) + cost$
                $p(v) \leftarrow u$
                if $v \notin Q$ then
                    $Q$.insert($v$)
                end
            end
        end
    end
end
if $d(t) = \infty$ then
    return $\emptyset$
end
return $\{(u \rightarrow v, flow, cap, cost) \in H \mid d(v) = d(u) + cost$ and $d(v) \leq d(t)\}$
```

3. In **topological order**, it distributes flow from each vertex to its successors as evenly as possible, respecting capacity constraints.

4. In **reverse topological order**, it adjusts flows to ensure conservation at each vertex.

5. Finally, it updates the actual flows in H and checks if any edge was saturated.

4.4.3 Extracting Weights From Flow Network

After computing the maximum flow in network H, we need to extract the inferred profile P''_e for the optimized CFG G_{opt}. This is done by Algorithm 13, which translates the flow values back into edge frequencies.

Algorithm 11: Run flow augmentation

```
Input: Flow network $H$ with source $s$ and sink $t$, augmenting DAG $X$
Output: void

Function compute_augmenting_capacity($H, s, t$):
    Initialize $P_C \leftarrow \infty$
    Initialize $cur \leftarrow t$
    while $cur \neq s$ do
        $pred \leftarrow p(cur)$
        $edge \leftarrow (pred \rightarrow cur, flow, cap, cost) \in H$
        $P_C \leftarrow \min(P_C, edge.cap - edge.flow)$
        $cur \leftarrow pred$
    end
    return $P_C$

// Main process begins here
Initialize $P_C \leftarrow$ compute_augmenting_capacity($H, s, t$)
while $P_C > 0$ do
    if not augment_flow_along_dag($H, s, t, X$) then
        $cur \leftarrow t$
        // Update flow in $H$
        while $cur \neq s$ do
            $pred \leftarrow p(cur)$
            $edge \leftarrow (pred \rightarrow cur, flow, cap, cost) \in H$
            $edge.flow \leftarrow edge.flow + P_C$
            $edge.reverse.flow \leftarrow edge.reverse.flow - P_C$
            $cur \leftarrow pred$
        end
        return
    end
    $P_C \leftarrow$ compute_augmenting_capacity($H, s, t$)
end
```

The algorithm initializes P''_e with the known values from the partial profile P'_e. Then, for each edge $E = (src \rightarrow dst)$ in the CFG, it finds the corresponding flow network edge $F_E = (src_O \rightarrow dst_I, flow, cap, cost)$. The flow value from F_E is added to $P''_e[E]$, incorporating the inferred flow into the profile.

A special case handles self-loops ($src = dst$): the flow is only added if it is positive, which may cause the edge to have no information, although it's inferred by the algorithm. However, this prevents negative frequencies in the final profile, which would be semantically invalid.

Algorithm 12: Augmenting flow along DAG

```
Input: Flow network $H$ with source $s$ and sink $t$, augmenting DAG $X$
Output: Boolean indicating if augmentation succeeded
Compute $F$ as the maximum flow that can be pushed through $X$ considering flow
  splitting
Initialize $\phi \leftarrow$ mapping from vertices and edges to flow amounts
$\phi(s) \leftarrow F$; $\phi(v) \leftarrow 0, \forall v \in V(X) - \{s\}$
// Forward pass: distribute the maximum amount of flow
foreach $u$ in topological order of $V(X) - \{t\}$ do
    $aux_F \leftarrow \lceil \frac{\phi(u)}{\delta(u)} \rceil$
    foreach $(u \to v, flow, cap, cost) \in X$ do
        $\phi(u, v) \leftarrow \min(\phi(u), aux_F, cap - flow)$
        $\phi(v) \leftarrow \phi(v) + \phi(u, v)$
        $\phi(u) \leftarrow \phi(u) - \phi(u, v)$
    end
end
// Backward pass: ensure flow conservation
foreach $v$ in reverse topological order of $V(X)$ do
    foreach $(u \to v, flow, cap, cost) \in X$ do
        $aux \leftarrow \min(\phi(v), \phi(u, v))$
        $\phi(v) \leftarrow \phi(v) - aux$
        $\phi(u) \leftarrow \phi(u) + aux$
        $\phi(u, v) \leftarrow \phi(u, v) - aux$
    end
end
// Update flow in $H$
foreach $edge \leftarrow (u \to v, flow, cap, cost) \in X$ do
    $edge.flow \leftarrow edge.flow + \phi(u, v)$
    $edge.reverse.flow \leftarrow edge.reverse.flow - \phi(u, v)$
end
return $\exists (u \to v, flow, cap, cost) \in X$ with $flow = cap$
```

4.4.4 Joining Isolated Components

Even after running the MCF algorithm, some basic blocks may remain unreachable from the entry block in the inferred profile. We define a basic block bb as *reachable* if there exists a path W from the entry block to bb where every edge $(src \to dst) \in W$ has $P''_e[(src \to dst)] > 0$.

If an unreachable block bb is incident to an edge with a positive profile edge (i.e. there exists an edge $(src \to dst)$ where bb is either src or dst and $P''_e[(src \to dst)] > 0$), we must connect it to the entry block. Algorithm 14 addresses this by finding and augmenting paths to these isolated components. The algorithm proceeds as follows:

Algorithm 13: Extracting weights from flow network.

```
Input: Optimized program $G_{opt}$, partial profile $P'_e$, flow network $H$
Output: Inferred profile $P''_e$

Initialize $P''_e \leftarrow P'_e$
foreach $edge\ E = (src \rightarrow dst) \in G_{opt}$ do
    $F_E = (src_O \rightarrow dst_I, flow, cap, cost) \in H$
    if $src \neq dst$ then
        $P''_e[E] \leftarrow P''_e[E] + flow$
    else
        if $flow > 0$ then
            $P''_e[E] \leftarrow P''_e[E] + flow$
        end
    end
end
return $P''_e$
```

1. It identifies all unreachable blocks using a graph traversal (e.g., BFS or DFS) starting from the entry block, considering only edges with positive flow.

2. For each unreachable block *bb* that is connected to an edge with positive profile value, it finds a connecting path:

   - A path from the entry block to *bb*
   - A path from *bb* to any exit block

3. These paths are found using Dijkstra's algorithm [13], where edge weights are inversely proportional to their profile values (higher profile $\Rightarrow$ lower weight). This preference for high-frequency paths helps maintain the profile's characteristics.

4. The algorithm then increments the profile of each edge along the combined path by 1, making *bb* reachable.

4.4.5 Rebalancing Undefined Subgraphs

The final phase of Profi [23] addresses basic blocks that remain without assigned profiles. These typically form acyclic subgraphs where only the root has a known profile, and internal blocks have undefined profiles. The algorithm identifies these subgraphs and distributes the root's frequency through them while maintaining flow conservation. Algorithm 15 coordinates this process:

Algorithm 14: Joining Isolated Components

```
Input: Optimized program $G_{opt}$, inferred profile $P''_e$
Output: Updated profile $P''_e$ without isolated components
Initialize $unreachable \leftarrow$ find_unreachable($G_{opt}$.entry)
foreach $basic\ block\ bb \in unreachable$ do
    if $\exists(src \rightarrow dst) \in G_{opt} \mid (src == bb \vee dst == bb)$ and $P''_e[(src \rightarrow dst)] > 0$
     then
        $path_{in} \leftarrow$ shortest_path($G_{opt}$.entry, $bb$, $P''_e$)
        $path_{out} \leftarrow$ shortest_path($bb$, $G_{opt}$.exits, $P''_e$)
        $path \leftarrow path_{in} \cup path_{out}$
        foreach $edge\ (u \rightarrow v) \in path$ do
            $P''_e[(u \rightarrow v)] \leftarrow P''_e[(u \rightarrow v)] + 1$
            if $v \in unreachable$ then
                $unreachable \leftarrow unreachable \setminus \{v\}$
            end
        end
    end
end
return $P''_e$
```

1. Compute the block profile P''_b from the edge profile P''_e
2. For each block bb with positive profile that has successors with undefined profiles:
    - Find the potential undefined subgraph rooted at bb (Algorithm 17)
    - Validate the subgraph structure (Algorithm 18)
    - Rebalance profiles within the valid subgraph (Algorithm 19)
3. Return the full profile as the union of the edge profile and the block profile

Algorithm 15: Rebalance Undefined Subgraphs

```
Input: Optimized program $G_{opt}$, edge profile $P''_e$
Output: Inferred profile $P''$
Use $P''_e$ and Equation 2.1 to build $P''_b$
foreach $basic\ block\ bb \in G_{opt}$ do
    if $P''_b[bb] > 0$ and $\exists(bb \rightarrow dst)$ with $P''_b[dst] = undef$ then
        $SG \leftarrow$ find_undefined_subgraph($G_{opt}$, $P''_b$, $P''_e$, $bb$)
        if valid_subgraph($G_{opt}$, $P''_b$, $SG$) then
            $P''_b, P''_e \leftarrow$ rebalance_subgraph($G_{opt}$, $P''_b$, $P''_e$, $SG$)
        end
    end
end
return $P'' \leftarrow P''_b \cup P''_e$
```

4.4.5.1 Finding Undefined Subgraph

An undefined subgraph is a maximal subgraph $SG \subseteq G_{opt}$ where:

- The root has a defined profile $P''_b > 0$
- Internal vertices have undefined profiles ($P''_b = \texttt{undef}$)
- Leaf nodes may have defined profiles

The subgraph consists of the root node, a set of *known* blocks (with defined profiles), and a set of *unknown* blocks (with undefined profiles).

Algorithm 17 identifies such subgraphs using a BFS traversal from the candidate root. It uses Algorithm 16 to determine which edges belong to the subgraph. An edge ($src \to dst$) is included if:

- It has non-zero profile ($P''_e[(src \to dst)] \neq 0$)
- The destination block has non-zero profile ($P''_b[dst] \neq 0$)
- If dst has a defined profile, then src cannot be the root (we want internal nodes between the root and the known blocks)

During BFS traversal:

- Blocks with defined profiles are added to the `known` set
- Blocks with undefined profiles are added to the `unknown` set and the BFS queue

Algorithm 16: Check if Edge is Considerable for Subgraph Finding

```
Input: Optimized program $G_{opt}$, inferred profiles $P''_b, P''_e$, root $root$, edge
       ($src \to dst$)
Output: Boolean indicating edge inclusion
if $P''_e[(src \to dst)] = 0$ or $P''_b[dst] = 0$ then
    return false
end
if $P''_b[dst] \neq undef$ and $src = root$ then
    return false
end
return true
```

Algorithm 17: Find Undefined Subgraph

```
Input: Optimized program $G_{opt}$, inferred profiles $P''_b, P''_e$, subgraph root $root$
Output: Undefined subgraph $SG = \{root, known, unknown\}$

Initialize $known \leftarrow \emptyset$, $unknown \leftarrow \emptyset$
Initialize $Q \leftarrow$ queue containing $root$
Initialize $Vis \leftarrow \{root\}$
while $Q$ not empty do
    $bb \leftarrow Q$.pop() foreach $edge$ $(bb \rightarrow dst) \in G_{opt}$ do
        if $dst \notin Vis$ and consider_edge($G_{opt}, P''_b, P''_e, root, (bb \rightarrow dst)$) then
            $Vis \leftarrow Vis \cup \{dst\}$
            if $P''_b[dst] \neq undef$ then
                $known \leftarrow known \cup \{dst\}$
            else
                $unknown \leftarrow unknown \cup \{dst\}$
                $Q$.insert($dst$)
            end
        end
    end
end
return $\{root, known, unknown\}$
```

4.4.5.2 Checking Subgraph Validity

Before rebalancing a subgraph, we must verify its structural validity. Algorithm 18 implements this check. A subgraph SG is considered valid if it satisfies the following conditions, that ensure the subgraph has a valid structure for profile inference while maintaining the algorithm's efficiency:

- **Non-empty unknown set**: $SG.unknown \neq \emptyset$ (must contain blocks needing inference)
- **At most one known block**: $|SG.known| \leq 1$ (simplifies rebalancing)
- **Proper leaf structure**: If a known block exists, it must be the only leaf node
- **Connected unknown blocks**: Every unknown block must have at least one outgoing edge with $P''_b[dst] \neq 0$, unless this block is a valid leaf node
- **Acyclic**: The subgraph must not contain cycles (simplifies rebalancing by allowing the extraction of a topological order)

Algorithm 18: Subgraph Validity

```
Input: Optimized program $G_{opt}$, inferred block profile $P''_b$, subgraph $SG$
Output: Boolean indicating subgraph validity
if $SG.unknown = \emptyset$ or $|SG.known| > 1$ then
    return false
end
foreach basic block $bb \in SG.unknown$ do
    if $bb$ has no outgoing edges in $G_{opt}$ then
        if $|SG.known| = 1$ then
            return false
        end
    else
        if $\forall edge(bb \to dst) \in G_{opt}, P''_b[dst] = 0$ then
            return false
        end
    end
end
if $SG.unknown$ contains a cycle then
    return false
end
return true
```

4.4.5.3 Rebalancing Subgraph

After validating a subgraph, Algorithm 19 redistributes profile to maintain the flow conservation, which happens in two phases:

1. **Root rebalancing**: Compute the root's frequency as the sum of its outgoing edges within the subgraph, then distribute it to successors.
2. **Unknown blocks rebalancing**: Process unknown blocks in topological order, where each block receives frequency from its incoming edges and distributes it to outgoing edges. The topological order ensures that, when processing a block *bb*, all the predecessors of *bb* have already been processed. Also, since the subgraph is acyclic, it is guaranteed that a topological order exists.

This algorithm uses 2 auxiliary algorithms: Algorithm 20, to identify edges belonging to the subgraph during rebalancing; and Algorithm 21, to distribute frequencies at individual blocks.

Algorithm 20 extends Algorithm 16 with an additional condition: edges leading to the subgraph's leaf node (if one exists) are always included. This ensures the rebalancing accounts for the subgraph's exit point.

Algorithm 19: Rebalance Subgraph

```
Input: Optimized program $G_{opt}$, profiles $P''_b, P''_e$, subgraph $SG$
Output: Updated profiles $P''_b, P''_e$
if $|SG.known| = 1$ then
    $SG.leaf \leftarrow$ the single known block in $SG.known$
else
    $SG.leaf \leftarrow \emptyset$
end
Initialize $frequency \leftarrow 0$
foreach edge $(SG.root \rightarrow dst) \in G_{opt}$ do
    if consider_edge($G_{opt}, P''_b, P''_e, SG.root, SG.leaf, (SG.root \rightarrow dst)$) then
        $frequency \leftarrow frequency + P''_e[(SG.root \rightarrow dst)]$
    end
end
$P''_e \leftarrow$ rebalance_block($G_{opt}, P''_b, P''_e, SG, SG.root, frequency$)
foreach basic block bb in topological order of $SG.unknown$ do
    $frequency \leftarrow 0$
    foreach edge $(src \rightarrow bb) \in G_{opt}$ do
        $frequency \leftarrow frequency + P''_e[(src \rightarrow bb)]$
    end
    $P''_b[bb] \leftarrow frequency$
    $P''_e \leftarrow$ rebalance_block($G_{opt}, P''_b, P''_e, SG, bb, frequency$)
end
return $P''_e, P''_b$
```

Algorithm 20: Check if Edge is Considerable for Subgraph Rebalancing

```
Input: Optimized program $G_{opt}$, inferred profiles $P''_b, P''_e$, root $root$, leaf $leaf$,
       edge $(src \rightarrow dst)$
Output: Boolean indicating edge inclusion
if $P''_e[(src \rightarrow dst)] = 0$ or $P''_b[dst] = 0$ then
    return false
end
if $dst = leaf$ then
    return true
end
if $P''_b[dst] \neq undef$ and $src = root$ then
    return false
end
return true
```

The block rebalance happens by distributing its frequency as evenly as possible to its successors within the subgraph. The process is very similar to the one that happens in Algorithm 12.

Algorithm 21: Rebalance Block

```
Input: Optimized program $G_{opt}$, profiles $P''_b, P''_e$, subgraph $SG$, block $bb$,
       frequency $frequency$
Output: Updated inferred edge profile $P''_e$
Initialize $\delta'(bb) \leftarrow 0$
foreach $edge\ (bb \rightarrow dst) \in G_{opt}$ do
    if consider_edge($G_{opt}, P''_b, P''_e, SG.root, SG.leaf, (bb \rightarrow dst)$) then
        $\delta'(bb) \leftarrow \delta'(bb) + 1$
    end
end
if $\delta'(bb) > 0$ then
    $aux \leftarrow \lceil \frac{frequency}{\delta'(bb)} \rceil$
    foreach $edge\ (bb \rightarrow dst) \in G_{opt}$ do
        if consider_edge($G_{opt}, P''_b, P''_e, SG.root, SG.leaf, (bb \rightarrow dst)$) then
            $f' \leftarrow \min(aux, frequency)$
            $P''_e[(bb \rightarrow dst)] \leftarrow f'$
            $frequency \leftarrow frequency - f'$
        end
    end
end
return $P''_e$
```

4.5 Projection with a Generative Model

This projection heuristic uses the same setup discussed in Section 3.3; however, inputs and prompt vary. The prompt is produced with information specific to each function that is analyzed and follows the template seen in Figure 4.7. The remainder of this section describes the methodology to generate this prompt.

Inputs. For each function, a script collects three textual inputs:

- The LLVM IR of the G_{ref} function compiled at -O0.
- Its basic block profile (`.bb`) and/or edge profile (`.edges`), if available.
- The LLVM IR of the optimized function G_{opt}.

Prompt Construction. The prompt builder script checks which types of profile (`.bb` or `.edges`) are available. Then it prepares a structured version of the prompt that asks the LLM to:

- Identify the hottest basic block in the optimized function.

```
Role: You are a Compiler Engineer with over 20 years of experience
in compiler construction, LLVM internals, and advanced optimization techniques.
You have deep expertise in low-level code generation, IR transformations, and
performance tuning for modern architectures, with a strong track record in static analysis,
JIT compilation, and custom backend development.

Task: You are given a function that exists in two versions:
- Its original LLVM IR at -${inOptLevel}: {iNdef}
- Its basic block counts (.bb profile)
- Its edge-level profile (.edges profile)
- Its optimized LLVM IR at ${optLevel}: {oNdef}

Your goal is to project the profile information from the -${inOptLevel} version onto the
optimized version. I have two questions regarding this function:

Question 1, Hot Block Estimation:

Could you guess which basic block would be the "hottest",  where "hottest" means the basic
block that is more likely to be executed more often?

Question 2, Hotness Ranking:

The optimized function has ${numBB} basic blocks: ${bbList}. Could you sort this sequence of
basic blocks by "hotness"? That is, if a basic block bb_x appears ahead of another block
bb_y, it means you think bb_y will not be executed more often than bb_x, though they might
have equal frequency.

Instructions:
1. Please respond using the format above.
2. Ensure that the “Sorted Basic Blocks by Hotness” section includes every one of the $
   {numBB} basic blocks listed in ${bbList}, without omission or duplication.**
3. After listing the sorted blocks, please provide a summary line confirming that the total
   number of blocks in your list is exactly ${numBB}.
4. If any block from ${bbList} is missing or duplicated in your sorted list, please
   explicitly acknowledge it and correct the list before finalizing your answer.
5. Do not repeat or fully echo the entire function. Focus on analysis. You may refer to
   specific lines or blocks but avoid copying the whole code.
```

Figure 4.7: Prompt template used to request a profile projection from GPT-4o. The template asks the model to identify the hottest basic block and to produce a ranking of blocks by hotness, along with reasoning, given an ordering of a previous version of the program.

- Sort all basic blocks of the optimized function by projected hotness.

Model Output. The LLM responds with:

- The name of the hottest block in the optimized function.
- A complete ranking of blocks by projected hotness.
- Notes explaining how it mapped blocks from -O0 to the optimized version, along with the computed O0 frequencies.

Validation and Post-Processing. The script converts the LLM's output into a structured JSON format. It also validates the ranking by checking if any block is missing, duplicated, or hallucinated. Finally, the script saves the JSON files per-benchmark and per-function that summarize the projection. This JSON output schema contains the following fields:

- **benchmarkName** — The benchmark name, derived from the file name.

- **funcName** — The analyzed function.
- **numBB** — Number of basic blocks in the optimized function.
- **originalSet** — The ground-truth set of basic blocks extracted from the optimized LLVM IR.
- **predictedSet** — The LLM-projected hotness ranking of basic blocks.
- **tokenCount** — Number of tokens used for the LLM prompt and response.
- **discrepancy** — A flag indicating whether the predicted set mismatches the ground truth.
- **duplicates** — List of duplicate block names produced by the LLM, if any.
- **missing** — Blocks that exist in the ground truth but were not included in the prediction.
- **extra** — Blocks hallucinated by the LLM that do not exist in the optimized IR.
- **exceed** — A flag indicating whether the prompt exceeded the token limit of the LLM.

Chapter 5

Experimental Evaluation

This chapter investigates several research questions concerning the heuristics introduced in Chapters 3 and 4. For clarity, we group these heuristics into two categories:

- **Classic**: comprising the random order (Random) from Section 3.1, the LLVM predictor (LLVM) from Section 3.2, the hash-based matching (Hash) from Section 4.2 and the histogram-based matching (Histogram) from Section 4.3.
- **LLM-Based**: comprising the techniques of prediction with a generative model (LLM-Pred), described in Section 3.3 and projection with a generative model (LLM-Proj), described in Section 4.5.

This differentiation is done specially because the *LLM-based* heuristics may not yield valid results in every benchmark, hence forcing the use of a subset of the benchmarks when they are compared. Within this division, we evaluate the following research questions:

RQ1: What is the relative accuracy of the classic prediction heuristics?

RQ2: What is the relative accuracy of the classic projection heuristics?

RQ3: How do the classic prediction heuristics compare with the LLM-based predictor?

RQ4: How do the classic projection heuristics compare with the LLM-based projector?

RQ5: How do different optimizations impact the accuracy of classic projection heuristics?

RQ6: What is the running time of the different classic prediction and projection heuristics?

RQ7: What is the running time per benchmark of the classic projection heuristics?

Evaluation Metric. We analyze the destructive effects that LLVM optimizations, such as constant global value numbering and loop rotation, may have on the quality of profile data. However, we cannot directly measure the potential benefits of profile propagation on these optimizations individually, because these optimizations, in LLVM, do not query profile information. Therefore, we resort to the Hot-Order Problem of Definition 2.4 as our evaluation metric. Accordingly, the accuracy of a heuristic on a benchmark is measured by comparing the block ordering it produces against a reference ordering obtained from the profile, that is obtained by running the instrumented program[1]. Following Definition 2.4, our goal is to minimize the minimum swap distance between the reference ordering R_P and the ordering predicted by the heuristic, H_P. We hence adopt the following definition of *accuracy*:

Definition 5.1 (Heuristic accuracy). *Let P be a program with N basic blocks, let R_P be the reference block ordering, and let H_P be the ordering predicted by a heuristic H. Accuracy is defined as the inversion of the distance between the minimum swap distance and the number of permutations that an array with N elements can have, which is the maximum value that a minimum swap distance can take. This is encoded by the following formula:*

$$Acc = 1 - \frac{2 \times \texttt{swap\_distance}(R_P, H_P)}{N \times (N-1)}$$

The resulting value $Acc \in [0,1]$, where 1 denotes a perfect match with the reference ordering.

Benchmarks. We evaluate the heuristics using the cBench benchmark suite [20], a collection of programs designed to assess profile-guided optimizations. The suite contains 32 programs, each with 20 distinct inputs. We successfully built and executed 17 of these programs; the remaining 15 were excluded due to compilation errors with modern `clang`, due to the benchmark being too obsolete, or errors during the use of the instrumentation framework. The way the benchmark programs and inputs are used vary in each research question, hence a brief explanation is done in the *methodology* paragraph of them. The results for the classic and LLM-based heuristics are based on slightly different subsets of benchmarks, as the LLM-based approaches described in Sections 3.3 and 4.5 did not yield valid results for every instance of the Hot-Order Problem. Consequently, Sections 5.3 and 5.4 restrict discussion to the subset of cBench for which the LLMs produced valid block sequences. A sequence is considered valid if it includes exactly all basic blocks from the program's control-flow graph.

Hardware and Software. Experiments were conducted on an Intel Core i7-6700T processor with 8 GB of RAM, two L1 caches (128 KB each), one L2 cache (1 MB), and one L3 cache (8 MB) running Ubuntu Linux 22.04.1 LTS. The heuristics were implemented

[1]The instrumented program is generated by a instrumentation-based profiler, described in the Hardware and Software paragraph

in LLVM 18.1.8. For profile collection, we used the Nisse framework [18][2], built from commit `25d9adf` of the main branch.

5.1 RQ1 – Accuracy of Classic Profile Prediction Heuristics

This section compares the accuracy of the classic profile prediction heuristics: Random order (*Random*) and LLVM predictor (*LLVM*). The goal here is to solve the Static Profile Prediction problem defined in Definition 2.5; in other words, profile data is reconstructed without any prior knowledge of the program's execution.

Methodology. We evaluate the heuristics from Sections 3.1 and 3.2 on the aggregate collection of functions from the 17 cBench programs that `clang` successfully compiles and that execute without errors across all 20 available inputs. The reported results correspond to the average accuracy (Definition 5.1) computed over all 168 functions from these benchmarks, with an average of 42 blocks per function, each executed with every input.

Heuristic	**-O0**	**-O1**	**-O2**	**-O3**
LLVM (Sec. 3.2)	**79.52%**	**77.73%**	**77.09%**	**77.09%**
Random (Sec. 3.1)	50.37%	50.15%	48.44%	50.50%

Table 5.1: Relative accuracy (Def. 5.1) of classic prediction heuristics. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks

Discussion. Table 5.1 presents the prediction accuracy obtained with four different optimization levels of `clang`. The results are consistent across levels. As expected, the Random heuristic achieves around 50% accuracy, which is explained by the following lemma:

Lemma 5.1. *Let $N \geq 1$ and consider the set of all permutations of N distinct elements. If a permutation is sampled uniformly at random, the expected accuracy, according to Definition 5.1, equals to* 50%*.*

Proof. We know that, by Theorem 3.1, that the expected minimum swap distance for a permutation of N distinct elements sampled uniformly at random equals to $\frac{N\times(N-1)}{4}$. So,

[2]The Nisse framework is a instrumentation-based profiler with an optimization to substitute some counters in the complement of the MST by affine variables

using this value in the formula of Definition 5.1, we have:

$$Acc = 1 - \frac{2 \times \frac{N\times(N-1)}{4}}{N \times (N-1)} = 1 - \frac{N \times (N-1)}{2 \times N \times (N-1)} = 1 - \frac{1}{2} = \frac{1}{2}$$

Which proves the lemma. □

The LLVM heuristic, on the other hand, maintains stable accuracy near 78–80%, indicating that it remains applicable throughout the optimization pipeline, even as transformations substantially alter the program structure.

However, this heuristic never exceeds 80% accuracy because it relies primarily on a fixed set of static branch prediction rules, such as loop-back edges being likely taken, pointer comparisons being likely unequal, and error paths being unlikely. These rules are general-purpose and context-insensitive: they capture common control-flow patterns but ignore dynamic interactions between branches, input-dependent behaviors, and correlations across basic blocks. Consequently, they can only approximate the true execution frequency of branches and basic blocks. We observe that, at least in our setting, such static heuristics reach a ceiling under 80% of accuracy. In the next section, we show that higher accuracy can be achieved when profile data from a different version of the optimized program is available.

5.2 RQ2 – Accuracy of Classic Profile Projection Heuristics

This section compares the classic projection heuristics discussed in Section 4: Hash-based matching (*Hash*) and histogram-based matching (*Histogram*). The goal is to solve the Profile Projection Problem introduced in Definition 2.6, that is, reconstruct the profile data of an optimized program given the profile and structure of the reference program, but without running the optimized program.

Methodology. For the projection, we get the profile from the reference program by instrumenting it with one input, and test the projection on the optimized program for all 20 inputs. To compute accuracy, we follow the same methodology used in Section 5.1, aggregating the average accuracy observed for each of the 168 functions across all 17 benchmarks and 20 input sets, with an average of 42 blocks per function.

Discussion. Table 5.2 summarizes our observations. In general, projection tends to outperform prediction, as shown by comparing Tables 5.1 and 5.2, although not in every setting. When the difference between program versions becomes too large, projection accuracy may degrade. This situation occurs when one of the program collections corresponds to the `-O0` optimization level. As we demonstrate in Section 5.5, this limitation is not problematic if profile data is propagated incrementally throughout the optimization pipeline; that is, from one optimization level to the next, rather than directly from an unoptimized to a fully optimized version of the same program.

	Heuristic	-O0	-O1	-O2	-O3
-O0	**Hash** (Sec. 4.2)	89.90%	65.28%	64.71%	63.95%
	Histogram (Sec. 4.3)	**95.55%**	**74.19%**	**73.63%**	**72.52%**
-O1	**Hash**	64.51%	91.60%	83.29%	81.57%
	Histogram	**71.01%**	**97.17%**	**89.36%**	**87.61%**
-O2	**Hash**	64.44%	83.55%	91.78%	89.55%
	Histogram	**71.73%**	**90.14%**	**96.95%**	**94.96%**
-O3	**Hash**	63.64%	81.34%	89.05%	91.19%
	Histogram	**70.51%**	**88.02%**	**94.76%**	**96.85%**

Table 5.2: Relative accuracy (Def. 5.1) of classic profile projection heuristics. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks.

The histogram-based matching heuristic of Section 4.3 consistently outperforms the hash-based matching in every configuration. A more detailed analysis of this result is provided in Section 5.5. In short, one of the main reasons for the underperformance of the Hash heuristic lies in its limited robustness to structural transformations such as loop rotation (illustrated in Figure 2.3) and optimizations that duplicate control-flow paths, including vectorization (which requires aliasing guards) and loop unrolling (which introduces an epilogue to handle residual iterations). These transformations alter the correspondence between basic blocks in the original and optimized programs, reducing the effectiveness of approaches that rely solely on hierarchical hash matching.

5.3 RQ3 – Comparison with LLM-Based Profile Prediction

This section compares the LLM-based profile predictor (*LLM-Pred*) described in Section 3.3 against the classic heuristics presented in Sections 3.1 and 3.2. As previously discussed, the number of functions had to be reduced because the generative model occasionally failed to produce valid orderings.

Methodology. The comparison is conducted over 127 functions (a subset of the total 168) extracted from the 17 benchmarks, with an average of 20 blocks per function, using every input. Invalid sequences either contained fewer or more basic blocks than the original control-flow graph, leading to an inconsistent block mapping. As in Section 5.1, the reported results correspond to the average accuracy computed over the benchmarks.

Discussion. Table 5.3 summarizes the results of this comparison. We observe that, although the LLM-based predictor clearly outperforms the random baseline, it consistently underperforms the LLVM heuristic, sometimes by a substantial margin. This gap can be attributed to the fundamental difference between the two approaches. LLVM's heuristic encodes domain-specific knowledge about compiler control flow, exploiting structural regularities such as loop backedges, branch biases, and dominance relations that were empirically tuned for real-world programs. By contrast, the LLM-based predictor operates purely on statistical correlations learned from textual representations of control-flow graphs.

Heuristic	**-O0**	**-O1**	**-O2**	**-O3**
LLM-Pred	69.81%	70.96%	69.84%	71.25%
LLVM	**81.53%**	**82.61%**	**80.30%**	**81.51%**
Random	50.22%	49.68%	47.97%	50.78%

Table 5.3: Comparison between classic prediction heuristics and the LLM-based predictor. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks.

Without explicit exposure to dynamic execution feedback, the model tends to generalize across unrelated contexts, producing plausible but not necessarily accurate block orderings. In particular, it often fails to capture the asymmetric frequency of paths within loops and conditionals. Our impression is that, when confronted with conditionals that generate multiple paths, the GPT-based model tends to select among them almost at random. Nevertheless, it appears capable of recognizing function entry and exit points, as it almost always puts these blocks at the end of the hot sequence. Hence, while LLM-based methods show promise, especially in settings that lack traditional compiler heuristics, they still fall short of outperforming domain-informed techniques.

5.4 RQ4 – Comparison with LLM-Based Profile Projection

This section compares the LLM-based profile projector (*LLM-Proj*) described in Section 4.5 with the classic heuristics presented in Section 4. The universe of functions end up being smaller than that of Section 5.3, since the prompts required for projection are substantially larger. The larger the prompt, the higher the likelihood that the generative model produces malformed or incomplete outputs.

Methodology. Like in Section 5.2, we use one input for collecting the profile, and all the 20 inputs for testing the projection method. As in Section 5.3, we use a reduced collection of 120 functions drawn from 17 benchmarks, with an average of 18 blocks per function, for the reasons discussed in our experimental setup. These prompts include two control-flow graphs, as illustrated in Figure 4.7.

Discussion. Table 5.4 summarizes our observations. We first note that LLM-based projection achieves higher accuracy than LLM-based prediction—an expected outcome, as the model receives additional information in the form of an ordered sequence of basic blocks from one version of the program. Nevertheless, the accuracy of the LLM-based approach still falls short of the other projection heuristics discussed in Sections 4.2 and 4.3. The underlying causes are similar to those discussed in Section 5.3: the absence of explicit semantic guidance and the model's tendency to rely on surface-level structural patterns rather than control-flow semantics.

We observe that the generative model often reproduces the hot-order sequence of the reference control-flow graph G_{ref} as the ordering for the optimized graph G_{opt}. When the differences between G_{ref} and G_{opt} are small, this strategy yields good accuracy. However, as structural differences grow, performance deteriorates sharply. In particular, when G_{opt} introduces new conditionals or merges existing paths not present in G_{ref}, the model appears to make random decisions among the alternative branches. This suggests that, while the LLM captures some high-level structural correspondences between graphs, it lacks the causal understanding of control-flow transformations that traditional, semantics-aware heuristics exploit.

	Heuristic	**-O0**	**-O1**	**-O2**	**-O3**
	Hash	94, 30%	67, 57%	65, 48%	64, 08%
-O0	**Histogram**	**95,34%**	**78,63%**	**77,80%**	**75,94%**
	LLM-Proj	80, 70%	72, 95%	73, 57%	72, 15%
	Hash	66, 02%	97, 92%	91, 05%	88, 14%
-O1	**Histogram**	**71,58%**	**98,17%**	**93,12%**	**90,49%**
	LLM-Proj	70, 00%	84, 55%	83, 16%	81, 24%
	Hash	65, 65%	91, 08%	97, 99%	96, 70%
-O2	**Histogram**	**74,28%**	**93,68%**	**98,21%**	**97,53%**
	LLM-Proj	71, 01%	80, 87%	84, 14%	84, 00%
	Hash	63, 74%	88, 62%	95, 85%	97, 59%
-O3	**Histogram**	**71,59%**	**90,81%**	**96,73%**	**97,92%**
	LLM-Proj	70, 44%	80, 89%	83, 35%	84, 86%

Table 5.4: Comparison between classic projection heuristics and LLM-based projection. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks.

5.5 RQ5 – Impact of Different Optimizations on Profile Projection

To understand why the Histogram heuristic is outperformed by the LLVM heuristic in the `-O0` setup, we conducted an additional set of experiments. This analysis evaluates the individual impact of each LLVM optimization pass on the classic profile projection heuristics.

Methodology. LLVM includes hundreds of optimization passes, making exhaustive analysis impractical. Therefore, we restrict our study to the 35 optimizations identified by Silva et al. [50] as the most common in effective `clang` optimization sequences for code-size reduction. Although the benchmarks used performance optimization flags, the list of optimizations remain valid for this study. Unlike previous sections, this experiment uses a single input for testing, the same used for training, to isolate the effects of optimizations on the quality of the projected profile. The adopted methodology is as follows:

1. Use Nisse to generate the profile of the reference control-flow graph G_{ref} with a given input I to obtain the reference ordering R_r;

2. Apply a single optimization pass to G_{ref}, producing the optimized graph G_{opt};

3. Use either the hash-based matching or the histogram-based matching heuristics to reconstruct the profile of G_{opt}, obtaining the projected ordering H_p;

4. Use Nisse to generate the profile of G_{opt} with the same input I to obtain the actual ordering R_p;

5. Compute the accuracy between R_p and H_p according to Definition 5.1.

We intentionally restrict the analysis to a single input to eliminate sources of variability that could confound the effects of optimizations on projection accuracy. Each cBench program provides 20 numbered inputs; for this experiment, we consistently use the first input.

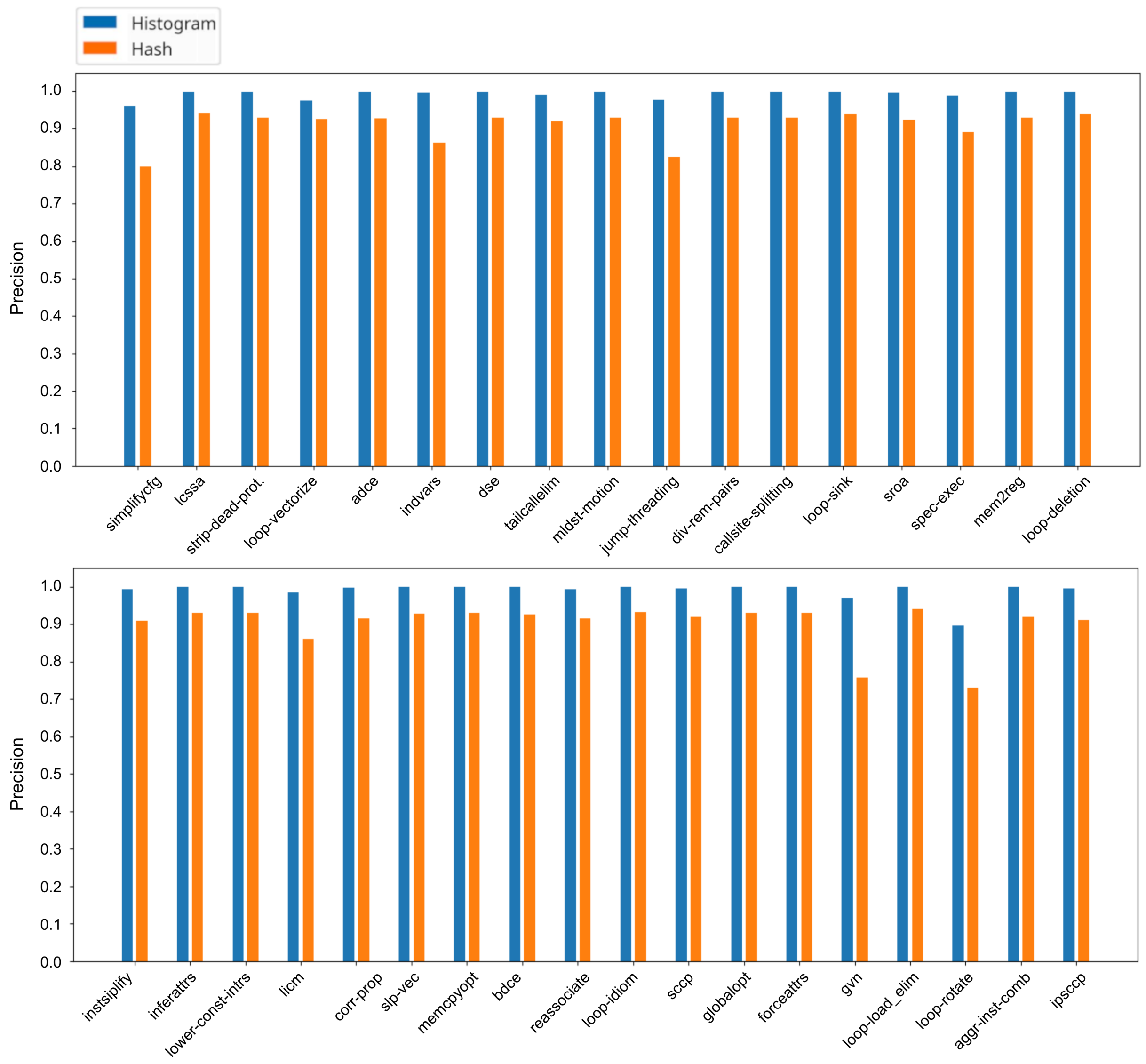


Figure 5.1: Impact of different optimizations on the accuracy of classic profile projection heuristics.

Discussion. Figure 5.1 presents the accuracy results for the Histogram heuristic (blue bars) and the Hash heuristic (orange bars). We observe that most optimizations have minimal impact on Histogram, whereas Hash is more sensitive to individual transformations. However, two optimization passes significantly reduce the accuracy of the histogram-based matching algorithm: `simplifycfg` and `loop-rotate`.

The `simplifycfg` pass alone does not drastically harm projection accuracy, with a precision of 96%; however, it is one of the most frequently applied transformations in the LLVM pipeline Silva et al.. This experiment isolates the standalone effect of each pass, but repeated applications of `simplifycfg` throughout a full optimization sequence could accumulate substantial distortion in the projected profiles.

Regarding `loop-rotate`, this optimization presents a significant challenge for both methods. It makes substantial changes to the CFG structure within loops. Although the loop-matching phase is designed to handle structural changes, it fails to fully capture the ones introduced by this optimization, which often creates new if-else structures to bypass the loop entry, thereby adding new basic blocks and edges.

5.6 RQ6 – Running Time of the Different Heuristics

This section compares the running time of the different *classic* heuristics evaluated in this paper. We do not report results for the LLM-based heuristics for two main reasons. First, their end-to-end execution time—from prompt submission to response generation—is orders of magnitude higher than that of the classic heuristics. Second, measuring this time accurately is technically challenging: the LLMs run on remote servers, and a precise measurement would require an infrastructure capable of accounting for variables such as network latency and server load.

Methodology. For the profile projection heuristics, we measure only the projection time itself, excluding the initial profile generation. The reported times correspond to the cumulative execution time of each heuristic across all 168 functions in the 17 benchmark programs.

Discussion. Table 5.5 reports the running times of the profile prediction heuristics, while Table 5.6 shows those of the projection heuristics. Profile prediction is substantially faster: the LLVM predictor runs roughly one order of magnitude faster than the hash-based matching, and two orders of magnitude faster than the histogram-based matching. In addition to algorithmic differences, this advantage arises from the fact that LLVM's predictor does not rely on solving a minimum-cost flow problem to reconstruct execution frequencies. The hash-based approach outperforms the histogram-based approach because it matches basic blocks in linear time using hash lookups, whereas the histogram-based matching performs pairwise similarity comparisons across blocks or loops, which is computationally more expensive. Nevertheless, the Histogram heuristic remains practical: it consistently completes within less than one minute for the entire universe of the 168 functions, regardless of the optimization level adopted.

Heuristic	**-O0**	**-O1**	**-O2**	**-O3**
LLVM	1.065s	1.434s	1.469s	1.531s
Random	0.612s	0.522s	0.505s	0.508s

Table 5.5: Execution time of classic profile prediction heuristics. Each row represents a heuristic. Each column represents the optimization flag used to compile the benchmarks

	Heuristic	**-O0**	**-O1**	**-O2**	**-O3**
-O0	**Hash**	13.752s	13.214s	8.641s	8.553s
	Histogram	22.284s	37.606s	42.480s	42.502s
-O1	**Hash**	8.672s	9.490s	8.363s	8.762s
	Histogram	42.972s	21.917s	43.724s	44.989s
-O2	**Hash**	8.999s	9.200s	9.286s	8.992s
	Histogram	47.266s	44.176s	23.428s	26.262s
-O3	**Hash**	9.010s	9.029s	9.479s	9.407s
	Histogram	47.912s	46.555s	27.912s	23.726s

Table 5.6: Execution time of classic profile projection heuristics. Each row represents the optimization level used to compile the reference program, from which we get the profile for projection, and then each row have 2 sub rows, one for each projection heuristic. Each column represents the optimization flag used to compile the benchmarks.

5.7 RQ7 – Running Time per Benchmark of the Classic Projection Heuristics

To better understand what benchmarks inflicts a higher running time between *Hash* and *Histogram*, we resort to a more specific experiment. Here, we specify what is the running time for each benchmark, as well as its size.

Methodology. We measure the running time of projecting the profile from a reference program optimized with `-O0` to a program optimized with `-O3`. We collect, as the size of each benchmark, the number of functions and the number of basic blocks. Like in Section 5.6, the reported time is the cumulative execution time of each heuristic across all functions in each benchmark.

Discussion. Table 5.7 shows the results of this experiment. We can see the linear and quadratic behavior of each algorithm, where a greater number of blocks increases significantly more the running time of *Histogram* in face of the *Hash* heuristic. Notice that the number of functions does not present a direct correlation with the running time, although it usually means more basic blocks. Another important thing to point out is that some functions can be very big in some rare cases, like the function `BZ2_decompress` from the benchmark **bzip2d**, with 793 basic blocks.

Benchmark	**# of functions**	**# of blocks**	**Hash**	**Histogram**
automotive_bitcount	11	144	0.07s	0.116s
automotive_qsort1	4	139	0.028s	0.095s
automotive_susan_c	18	906	0.558s	2.472s
automotive_susan_e	18	906	0.493s	2.285s
automotive_susan_s	18	906	0.611s	2.239s
bzip2d	46	4456	3.041s	17.064s
bzip2e	46	4456	2.895s	16.921s
network_dijkstra	5	90	0.033s	0.056s
network_patricia	6	164	0.075s	0.169s
office_stringsearch1	9	289	0.093s	0.15s
security_blowfish_d	6	145	0.134s	0.216s
security_blowfish_e	6	145	0.159s	0.223s
security_rijndael_d	8	169	0.193s	0.291s
security_sha	6	53	0.055s	0.052s
telecom_CRC32	4	62	0.02s	0.031s
telecom_adpcm_c	3	36	0.029s	0.067s
telecom_adpcm_d	3	36	0.065s	0.054s

Table 5.7: Execution time of classic profile projection heuristics per benchmark. Each row represents a different benchmark

Chapter 6

Related Work

This chapter presents a brief overview of the literature related to intra-procedural binary correspondence, which is closely related to the problem of projecting profile information from one version to another version of the same function. Also, the chapter presents different techniques for static profile prediction that were not covered in this work.

6.1 Binary Diffing

The term binary diffing is used in the programming literature with different interpretations. Different authors employ the concept to address distinct problems:

- Hemel et al. [24] uses binary diffing to determine whether two binaries originate from the same source code or different versions of the same source code.
- Ming et al. [36] focuses on determining whether two binaries implement similar semantics, i.e., whether they solve the same computational problem.
- Kim et al. [28] understands diffing as intra-procedural code correspondence, matching code elements within functions.

These divergent interpretations give two distinct approaches to the problem known as binary diffing [1, 22]:

- **Stochastic techniques:** employ machine learning models, such as that of Shin et al. [49], to solve the semantic problem of determining when two binaries execute similar actions.
- **Hash-based approaches:** align binary code by finding correspondences either between functions (inter-procedural correspondence) or between control-flow graphs (intra-procedural correspondence).

The profile projection techniques seen in Chapter 4 can be used to solve the hash-based version of the problem, as they operate at the control-flow graph level to establish intra-procedural correspondences.

6.2 Inter-Procedural Correspondence

In the words of Flake [16], the inter-procedural correspondence consists of "Given two variants of the same executable A called A' and A'', a one-to-one mapping between all the functions in A' to all the functions in A'' is created".

This approach typically extracts characteristics from each function and establishes correspondence based on these features. The used characteristics can be:

- **Structural:** like properties from the control-flow graph of the function, such as the number of nodes and edges, circumference, tree-width, etc. [35]
- **Semantical:** like instruction opcode histograms [12] and many vectors built as combinations of instructions from the program [14].

The main motivation for developing inter-procedural techniques is the need to deal with binary code without high-level information, like function names. However, industrial systems performing profile-guided optimization on large binaries typically have access to such high-level information [9, 43, 45, 52]. Notice that, in the context of this work, such correspondence does not apply, as we focus on intra-procedural analysis and do not deal with complete binary programs.

6.3 Intra-Procedural Correspondence

The branching mapping problem discussed in this chapter is an instance of intra-procedural binary correspondence. Many works deal with this problem, as discussed in the literature review done by Kim and Notkin [29]. In this case, the challenge is to find a correspondence between anchor points between two functions, with the anchor point type defining the solution nature. There are three main anchor point types commonly used:

- Nodes in abstract syntax trees [57], when source code is available

- Instructions in linear sequences of binary code [25]
- Vertices in control-flow graphs (The profile projection techniques seen in Chapter 4 fit in this category)

Common solutions for graph correspondence are based on graph isomorphism or sequence alignment. Graph isomorphism can be applied to both abstract syntax trees and control-flow graphs. Sequence alignment is usually solved with heuristics because of the binary programs size, through hash codes derived from instructions k-grams [3, 27].

Another line of research focuses on finding correspondences between functions using feature vectors extracted from them. These feature vectors can be hand-crafted from program characteristics, as in Moreira et al. [39], or learned automatically using machine learning algorithms, as in VenkataKeerthy et al. [53]. Although these approaches show promising results for static profile projection, the engineering required to either hand-craft the feature vectors or design and train a machine learning model makes them infeasible for the scope of this work.

6.4 Static Profile Prediction with Neural Networks

The static profile prediction problem can be addressed by applying machine learning to predict program behavior without execution. Čugurović et al. [59] used an XGBoost [10] model trained on manually engineered features from the compiler's intermediate representation. While effective, the manual feature extraction limited its ability to represent the program's structural complexity.

The GraalNN framework [34] represents an advancement by employing Graph Neural Networks that directly process control-flow graphs, inherently learning structural patterns and reducing the need for manual feature engineering. A key innovation is its two-stage process: it first predicts context-insensitive profiles, then refines them with context-sensitive information during inlining.

Both techniques are implemented within the GraalVM Native Image compiler, which prevents direct comparison with our work for several reasons. First, the community version of GraalVM lacks a profile-guided optimization implementation. Second, the approach is compiler-specific, with tight coupling with GraalVM's intermediate representation, which makes reproduction in frameworks like LLVM challenging. Finally, comparison with standard binary optimizers like BOLT is not possible due to architectural differences. Despite these differences, GraalNN shares our focus on control-flow graph analysis for profile inference, though it employs learned representations rather than the heuristic-based approaches we explore in Chapter 4.

Chapter 7

Conclusion

This work addressed a long-standing challenge in profile-guided optimization: keeping execution profiles accurate as the compiler or the developer transform code. We introduced a systematic study of two strategies for carrying profile information forward without rerunning the program: **prediction**, which estimates hot paths directly from optimized code, and **projection**, which transfers data from an earlier version of the program. Our evaluation shows that a simple **instruction histogram** heuristic consistently matches or outperforms more complex alternatives, including the hash-based method used in the BOLT Binary Optimizer, the heuristics used in LLVM's static profile, and large language model (LLM) techniques based on GPT-4o. Under standalone optimizations, the histogram-based matching heuristic achieves very high accuracy, suggesting the possibility of interposing it between consecutive optimizations in the compilation pipeline.

7.1 Future Work

Our work identified significant challenges in achieving high precision with general-purpose profile heuristics. As the evaluation in Chapter 5 demonstrates, the best precision for reconstructing a profile, either from an unoptimized program or without prior knowledge, remained below 80% for programs compiled with `-O1`, `-O2`, or `-O3`. This level of accuracy is insufficient for practical profile propagation. A more promising direction is to improve the projection of profile information through individual optimization passes within the compilation pipeline.

Compilation with PGO at Intermediate-Level Optimizations In this study, we introduced several heuristics for predicting and projecting profile information. Among them, the Histogram-Based approach (Section 4.3) proved particularly effective for propagating profiles through standalone optimizations. This suggests a promising venue for future work: improving existing intermediate-level optimizations through the use of profile information. A key challenge is determining when to regenerate profile data during compilation, as a sequence of transformations may render even a projected profile stale, degrading its utility for subsequent passes.

Improvement of LLM-Based Heuristics Although our LLM-based heuristics did not fare better than our histogram-based similarity search, we believe that these techniques offer exciting directions for future work. One possible path is to explore richer ways to support LLM-based profiling. In particular, it could be possible to provide the model with a compact but informative view of the LLVM IR for each function that might otherwise exceed the token window, enabling the LLM to speculate effectively even with partial data.

Optimization-Specific Profile Projection Heuristics Doing a perfect general purpose profile projection heuristic is likely impossible. A more feasible alternative is to create specialized heuristics tailored to the semantics of specific optimization passes. Such heuristics could leverage intrinsic knowledge of a pass's transformations, prioritizing high accuracy for that particular pass without caring for universal applicability. This approach could effectively address the challenges posed by passes like `loop-rotate` (discussed in Section 5.5), where general-purpose methods struggle.

Bibliography

[1] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, and Aiman Hanna. *Binary Analysis Overview*, pages 7–44. Springer International Publishing, Berlin, Heidelberg, 2020. ISBN 978-3-030-34238-8. doi: 10.1007/978-3-030-34238-8_2. URL https://doi.org/10.1007/978-3-030-34238-8_2.

[2] Amir Ayupov, Maksim Panchenko, and Sergey Pupyrev. Stale profile matching. In *Compiler Construction*, page 162–173, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705076. doi: 10.1145/3640537.3641573. URL https://doi.org/10.1145/3640537.3641573.

[3] Brenda S Baker, Udi Manber, and Robert Muth. Compressing differences of executable code. In *WCSS*, pages 1–10, New York, US, 1999. Citeseer Princeton, NJ, USA, ACM.

[4] Thomas Ball and James R. Larus. Branch prediction for free. In *PLDI*, page 300–313, New York, NY, USA, 1993. Association for Computing Machinery. ISBN 0897915984. doi: 10.1145/155090.155119. URL https://doi.org/10.1145/155090.155119.

[5] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994. ISSN 0164-0925. doi: 10.1145/183432.183527. URL https://doi.org/10.1145/183432.183527.

[6] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1): 87–90, 1958.

[7] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, jan 1997. ISSN 0164-0925. doi: 10.1145/239912.239923. URL https://doi.org/10.1145/239912.239923.

[8] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2): 376–389, 2013. doi: 10.1109/TC.2011.233.

[9] Dehao Chen, David Xinliang Li, and Tipp Moseley. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO*, page 12–23, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450337786. doi: 10.1145/2854038.2854044. URL https://doi.org/10.1145/2854038.2854044.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

[11] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *POPL*, page 70–85, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 9781450373470. doi: 10.1145/512644.512651. URL https://doi.org/10.1145/512644.512651.

[12] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintao Pereira, Nilton Queiroz, and Otavio Oliveira Napoli. Program representations for predictive compilation: State of affairs in the early 20's. *Journal of Computer Languages*, 73(101153): 101171, apr 2022. doi: 10.1016/j.cola.2022.101171.

[13] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959. ISSN 0945-3245. doi: 10.1007/BF01386390. URL https://doi.org/10.1007/BF01386390.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *SP*, pages 472–489, Washington, DC, USA, 2019. IEEE. doi: 10.1109/SP.2019.00003.

[15] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, April 1972. ISSN 0004-5411. doi: 10.1145/321694.321699. URL https://doi.org/10.1145/321694.321699.

[16] Halvar Flake. Structural comparison of executable objects. In Ulrich Flegel and Michael Meier, editors, *DIMVA*, volume P-46 of *LNI*, pages 161–173, Dortmund, Germany, 2004. GI. URL https://dl.gi.de/20.500.12116/29199.

[17] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi: 10.4153/CJM-1956-045-5.

[18] Leon Frenot and Fernando Magno Quintão Pereira. Reducing the overhead of exact profiling by reusing affine variables. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, CC 2024, page 150–161, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705076. doi: 10.1145/3640537.3641569. URL https://doi.org/10.1145/3640537.3641569.

[19] Elisa Fröhlich, Angélica Aparecida Moreira, and Fernando Magno Quintão Pereira. Automatic propagation of profile information through the optimization pipeline. In *Proceedings of the annual ACM conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2026. Accepted for presentation.

[20] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7(4), December 2011. ISSN 1544-3566. doi: 10.1145/1880043.1880047. URL https://doi.org/10.1145/1880043.1880047.

[21] T. Glek and J. Hubicka. Optimizing real world applications with gcc link time optimization, 2010. URL https://arxiv.org/abs/1010.2196.

[22] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), apr 2021. ISSN 0360-0300. doi: 10.1145/3446371. URL https://doi.org/10.1145/3446371.

[23] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. Profile inference revisited. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022. doi: 10.1145/3498714. URL https://doi.org/10.1145/3498714.

[24] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *MSR*, page 63–72, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305747. doi: 10.1145/1985441.1985453. URL https://doi.org/10.1145/1985441.1985453.

[25] He Huang, Amr M. Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *ASIA CCS*, page 155–166, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349444. doi: 10.1145/3052973.3052974. URL https://doi.org/10.1145/3052973.3052974.

[26] Teresa Louise JOHNSON and Xinliang David Li. Framework for user-directed profile-driven optimizations, September 12 2017. US Patent 9,760,351.

[27] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: A search engine for binary code. In *MSR*, page 329–338, New York, NY, USA, 2013. IEEE Press. ISBN 9781467329361.

[28] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. Improving cross-platform binary analysis using representation learning via graph alignment. In *ISSTA*, page 151–163, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393799. doi: 10.1145/3533767.3534383. URL https://doi.org/10.1145/3533767.3534383.

[29] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *MSR*, page 58–64, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933972. doi: 10.1145/1137983.1137999. URL https://doi.org/10.1145/1137983.1137999.

[30] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322, Sep 1973. ISSN 1572-9125. doi: 10.1007/BF01951942. URL https://doi.org/10.1007/BF01951942.

[31] Roy Levin, Ilan Newman, and Gadi Haber. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *HiPEAC*, page 291–304, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540775595.

[32] Bingxin Liu, Yinghui Huang, Jianhua Gao, Jianjun Shi, Yongpeng Liu, Yipin Sun, and Weixing Ji. From profiling to optimization: Unveiling the profile guided optimization, 2025. URL https://arxiv.org/abs/2507.16649.

[33] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. ISSN 1049-3301. doi: 10.1145/272991.272995. URL https://doi.org/10.1145/272991.272995.

[34] Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. Graalnn: Context-sensitive static profiling with graph neural networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 123–136, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712753. doi: 10.1145/3696443.3708958. URL https://doi.org/10.1145/3696443.3708958.

[35] Jiang Ming, Meng Pan, and Debin Gao. Ibinhunt: Binary hunting with interprocedural control flow. In *ICISC*, '12, page 92–109, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 9783642376818. doi: 10.1007/978-3-642-37682-5_8. URL https://doi.org/10.1007/978-3-642-37682-5_8.

[36] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *SEC*, page 253–270, USA, 2017. USENIX Association. ISBN 9781931971409.

[37] Edward F. Moore. The shortest path through a maze. In *Proc. Internat. Sympos. Switching Theory 1957, Parts I,II*, volume vols. XXIX, XXX of *The Annals of the Computation Laboratory of Harvard University*, pages 285–292. Harvard Univ. Press, Cambridge, MA, 1959.

[38] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: Static profiling for binary optimization. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485521. URL https://doi.org/10.1145/3485521.

[39] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Pereira. Beetle: A feature-based approach to reduce staleness in profile data, July 2023. URL https://doi.org/10.22541/au.168898868.83064146/v1.

[40] Ivan Murashko. *Clang Compiler Frontend: Get to grips with the internals of a C/C++ compiler frontend and create your own tools*. Packt Publishing, Birmingham, UK, 2024.

[41] Diego Novillo. Samplepgo: the power of profile guided optimizations without the usability burden. In Hal Finkel and Jeff R. Hammond, editors, *LLVM Compiler Infrastructure in HPC*, pages 22–28, New York, NY, USA, 2014. IEEE Computer Society. doi: 10.1109/LLVM-HPC.2014.8. URL https://doi.org/10.1109/LLVM-HPC.2014.8.

[42] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. Establishing a base of trust with performance counters for enterprise workloads. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 541–548, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/nowak.

[43] Guilherme Ottoni. HHVM JIT: A profile-guided, region-based compiler for PHP and Hack. In *PLDI*, page 151–165, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Maksim Panchenko. Optimizing clang : A practical example of applying BOLT. https://github.com/facebookincubator/BOLT/blob/master/docs/OptimizingClang.md, accessed on 2020-12-11, 2018.

[45] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A practical binary optimizer for data centers and beyond. In *CGO*, page 2–14, Washington, DC, USA, 2019. IEEE Press. ISBN 9781728114361. doi: 10.5555/3314872.3314876.

[46] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE*

Micro, 30(4):65–79, July 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.68. URL https://doi.org/10.1109/MM.2010.68.

[47] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73562. URL https://doi.org/10.1145/73560.73562.

[48] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *ASPLOS*, page 617–631, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575727. URL https://doi.org/10.1145/3575693.3575727.

[49] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *SEC*, page 611–626, USA, 2015. USENIX Association. ISBN 9781931971232.

[50] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. Exploring the space of optimization sequences for code-size reduction: insights and tools. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 47–58, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383257. doi: 10.1145/3446804.3446849. URL https://doi.org/10.1145/3446804.3446849.

[51] Julian Templeman. *Microsoft Visual C++/CLI Step by Step*. Pearson Education, London, UK, 2013.

[52] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. One profile fits all: Profile-guided linux kernel optimizations for data center applications. *SIGOPS Oper. Syst. Rev.*, 56(1):26–33, jun 2022. ISSN 0163-5980. doi: 10.1145/3544497.3544502. URL https://doi.org/10.1145/3544497.3544502.

[53] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul PS, Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. Vexir2vec: An architecture-neutral embedding framework for binary similarity. *ACM Trans. Softw. Eng. Methodol.*, March 2025. ISSN 1049-331X. doi: 10.1145/3721481. URL https://doi.org/10.1145/3721481. Just Accepted.

[54] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[55] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *MICRO*, page 1–11, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897917073. doi: 10.1145/192724.192725. URL https://doi.org/10.1145/192724.192725.

[56] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers. In *International Conference on Supercomputing*, ICS '19, page 284–295, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360791. doi: 10.1145/3330345.3330371. URL https://doi.org/10.1145/3330345.3330371.

[57] Wuu Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, jun 1991. ISSN 0038-0644. doi: 10.1002/spe.4380210706. URL https://doi.org/10.1002/spe.4380210706.

[58] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 98–105, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380690. doi: 10.1145/3409963.3410490. URL https://doi.org/10.1145/3409963.3410490.

[59] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Würthinger. Graalsp: Polyglot, efficient, and robust machine learning-based static profiler. *Journal of Systems and Software*, 213:112058, 2024. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2024.112058. URL https://www.sciencedirect.com/science/article/pii/S0164121224001031.