Automatic Propagation of Profile Information through the Optimization Pipeline

ELISA FRÖHLICH, UFMG, Brazil
ANGÉLICA APARECIDA MOREIRA, Microsoft, USA
FERNANDO M. QUINTÃO PEREIRA, UFMG, Brazil

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, rendering the collected profiles stale or inaccurate. To deal with this problem, this paper investigates how to reuse 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, 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.

CCS Concepts: • **Software and its engineering** → **Compilers**.

Additional Key Words and Phrases: Static Analysis, Profile Projection, Profile Prediction

ACM Reference Format:
Elisa Fröhlich, Angélica Aparecida Moreira, and Fernando M. Quintão Pereira. 2026. Automatic Propagation of Profile Information through the Optimization Pipeline. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2026), 26 pages. https://doi.org/00.0000/0000000

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 [14], LLVM/Clang [29], and Microsoft Visual C++ [38], and is routinely applied in large software systems [33, 35]. 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 [20]. This approach can yield substantial performance improvements, with speedups of up to 20–30% being reported in both academic studies and industrial practice [27, 28, 31, 32].

Profile Usage and Preservation: a Circular-Dependency Problem. 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 act of optimizing 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

Authors' Contact Information: Elisa Fröhlich, UFMG, Belo Horizonte, Minas Gerais, Brazil, elisa.frohlich@dcc.ufmg.br; Angélica Aparecida Moreira, Microsoft, Redmond, Seattle, USA, anmoreira@microsoft.com; Fernando M. Quintão Pereira, UFMG, Belo Horizonte, Minas Gerais, Brazil, fernando@dcc.ufmg.br.

update or repair profile information as transformations are applied. Concrete instances include the function `updateBranchWeights` in `LoopPeel.cpp`[1], which recalculates branch frequencies after loop peeling, and `updatePredecessorProfile- Metadata` in `JumpThreading.cpp`[2], 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[3], often resulting in large and intricate patches that require careful design and extensive review.

Profile Matching: Shortcomings of Current Approaches. There are techniques that attempt to project profile data from one version of a program onto another [2, 27, 40]. In principle, these methods could be used to propagate profile information along the optimization pipeline. However, the three techniques we are aware of were not designed with compiler optimizations in mind. Their primary goal is to enable the reuse of profile data during program development.

The approaches of Wang et al. [40] and Ayupov et al. [2] project a profile from an older version of a program P_0 onto a newer version P_1 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, Wang et al. and Ayupov et al. 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. [27] 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[4]. Nevertheless, as we will show in Section 5, exact matching, even with relaxed hash functions, leads to an excessive number of misses in the presence of compiler optimizations.

The Contribution of This Paper: Heuristics for Similarity Matching. This paper 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 paper 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':

- **Ayupov:** apply the hierarchy of hash functions proposed by Ayupov et al. [2] to match CFG edges (see Section 4.1).
- **HistLoop:** recursively compare opcode histograms of code regions using Euclidean distance as a similarity criterion to match CFG edges (see Section 4.2).
- **LLM-Proj:** use an LLM to infer S', given knowledge of P, P', and S (see Section 4.4).

[1]Available in https://llvm.org/doxygen/LoopPeel_8cpp_source.html on September 8th, 2025.
[2]Available in https://llvm.org/doxygen/JumpThreading_8cpp_source.html on September 8th, 2025.
[3]For a recent proposal, see https://discourse.llvm.org/t/rfc-profile-information-propagation-unittesting/73595
[4]Moreira et al.'s 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.*"

Some of these heuristics rely on matching criteria proposed in prior work, such as those by Ayupov et al. [2] (currently in use in the BOLT binary optimizer [32]). 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 paper addresses for the first time. Furthermore, the heuristics in Sections 3.3, 4.2 and 4.4 are novel.

Summary of Results. We have implemented the heuristics proposed in this paper within the LLVM compilation infrastructure (release 18.1.8). Section 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-4.0o 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 paper 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 *exactly*.

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

The Matching Game: To ease the task of comparing different profile prediction or profile project approaches, Section 2.1 introduces a simple profiling game: predicting the "hot-order" of basic blocks in a program's control-flow graph. This game simplifies the comparison of different heuristics, as it removes measurement fluctuations from accuracy reports.

Similarity Search: Section 4.2 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.

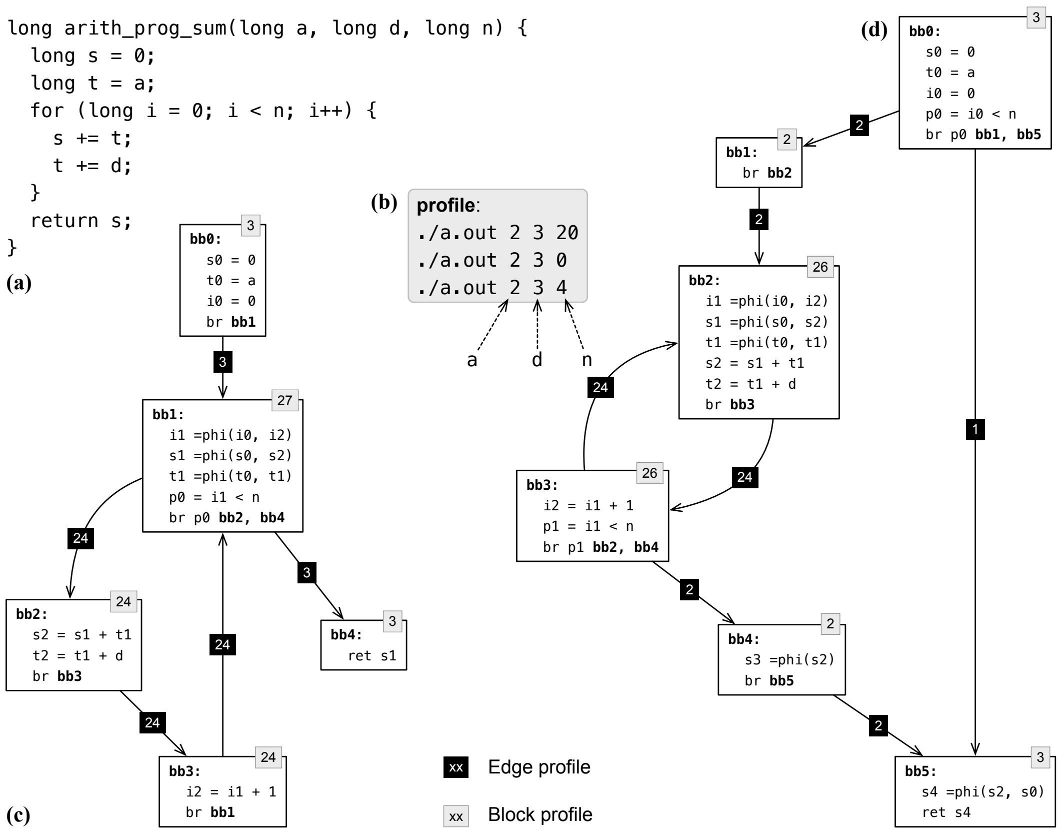
2 Profile Projection and Profile Prediction

The goal of this paper is to enable the use of profile information across compiler optimizations. We identify two approaches to achieving this goal: *prediction* of profile information and *projection* of profile information. This section defines both terms. Before doing so, however, we formalize the notion of a *program profile* (Definition 2.1) and illustrate it with an example (Example 2.2).

Definition 2.1 (Profile Information). The control-flow graph (CFG) of a program $G = (V, E)$ consists of a set of vertices V, representing basic blocks, and a set of directed edges $(u, v) \in E$, where an edge indicates that block v may immediately follow block u during execution. 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 must equal the block frequency, which in turn equals the sum of the frequencies of outgoing edges, except at entry and exit nodes:

$$\sum P_e(u \longrightarrow v) \;=\; P_b(v) \;=\; \sum P_e(v \longrightarrow w). \tag{1}$$

Example 2.2. Figure 1 (a) shows a function that computes the sum of an arithmetic progression iteratively. Part (b) depicts three different executions of a compiled version of this function. Part (c) presents the control-flow graph of the program,[5] annotated with profile information obtained from the executions. Finally, part (d) shows the CFG of the same program after loop rotation with profile data. This data comes from the three executions in Figure 1 (b).



```
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;
}
```

```
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
```

Fig. 1. (a) Function computing the sum of an arithmetic progression via iterative summation. (b) Three executions producing profile data. (c) Control-flow graph of the program with profile information. (d) Optimized CFG after loop rotation, with profile data carried forward.

Example 2.2 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. Such total profiles can be produced by instrumentation [12] or by sampling [15], complemented with techniques to fill in missing edges while respecting the conservation law of Equation 1, as implemented by BOLT [32]. Throughout this work we assume that program profiles are always total functions.

[5]We use a simplified version of the LLVM intermediate representation. It preserves the Single-Static Assignment form [7] but omits types and auxiliary metadata.

2.1 The Hot-Order Problem

We now introduce a benchmark for measuring the quality of a statically reconstructed profile. This benchmark, called the *Hot-Order Game*, 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 Game is defined as follows:

Definition 2.3 (The Hot-Order Game). 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 of V, where $v = L_i \in V$ denotes the i-th block in the sequence.
- **Goal:** Let $S_R = [P_b(v_1), \ldots, P_b(v_n)]$ 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., $S_L = [P_b(L_1), \ldots P_b(L_n)]$. Let D be the *Swap Distance*[6] between S_L and S_R. The objective of the Hot-Order Game is to minimize D.

Ordering blocks by their temperature serves as a benchmark for the accuracy of profile reconstruction techniques. In other words, Definition 2.3 can be used to evaluate: (i) methods that reconstruct execution frequencies of basic blocks [27]; (ii) approaches that complete missing information in sample-based profilers [2, 17]; or (iii) techniques that estimate branch probabilities [5, 28]. 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 2.2, a reference hot-ordering of the program in Figure 1(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 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`).

2.2 Static Profile Prediction

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:

- **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.

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 [41] can be used to reconstruct L. Alternatively, approaches that estimate branch probabilities [5, 28] are also applicable, since probabilities can be converted into execution frequencies [4].

2.3 Static Profile Projection

Static profile projection refers to techniques that attempt to map a profile produced for a reference control-flow graph onto another—optimized—version of that control-flow graph, without executing this new version. We formalize this problem as follows:

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

[6]The 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 Swap Distance is 1 (one swap between c and b).

- **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.

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.

3 Profile Reconstruction Heuristics

This section describes 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.

3.1 Random Order

This heuristic produces a random ordering of the basic blocks using the Mersenne Twister algorithm [24] as the random number generator. 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 a uniformly random ordering of the blocks. Theorem 3.1 (For the proof, see Section **??**) gives the expected swap-distance under that null model: $\mathbb{E}[\text{swap-distance}] = N(N-1)/4$. If a heuristic yields an average 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 adjacent-swap distance (i.e., the expected number of adjacent swaps required to transform the permutation into the sorted order, equivalently the expected number of inversions) equals*

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

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

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

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

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

By linearity of expectation,

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

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. Specifically, it relies on `BranchProbabilityInfoAnalysis`, an analysis pass that 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 Wu and Larus [41], such as: loop backedges are likely to be taken; error-handling paths are unlikely; equality tests (==) tend to evaluate to false; and so forth. Given these statically assigned probabilities, we compute a hot-order of basic blocks in three steps:

(1) **Initialization:** Assign frequency 1 to the entry block, assuming it is executed once.
(2) **Propagation:** Apply the Ball–Larus algorithm [4] to propagate execution frequencies throughout the CFG. Transition probabilities between blocks are taken from `BranchProbali-bityInfoAnalysis` which extends the original Ball–Larus method.
(3) **Ordering:** Sort the basic blocks according to the frequencies estimated in the previous step.

Example 3.2. Figure 2 illustrates how the LLVM predictor works on the program from Example 2.2(a). Part (a) shows the program after initialization, with frequency 1 assigned to the entry block, while the remaining blocks have undefined frequencies. Part (b) presents the probabilities between blocks as computed by `BranchProbabilityInfoAnalysis`. These probabilities are derived from LLVM's static heuristics: for instance, loop backedges are assumed to be very likely, which explains the high probability (0.96875) assigned to the edge returning to the loop header, while the exit edge receives only 0.03125. Part (c) shows the result of applying the Ball–Larus algorithm [4], which propagates the initial frequency through the CFG by multiplying it with the edge probabilities and ensuring flow conservation at each block. Because the backedge is taken with probability close to one, the loop header is expected to be visited many times, leading to an estimated edge frequency of 31 for the backedge. Finally, part (d) shows the block frequencies, computed from the edge frequencies using Equation 1.

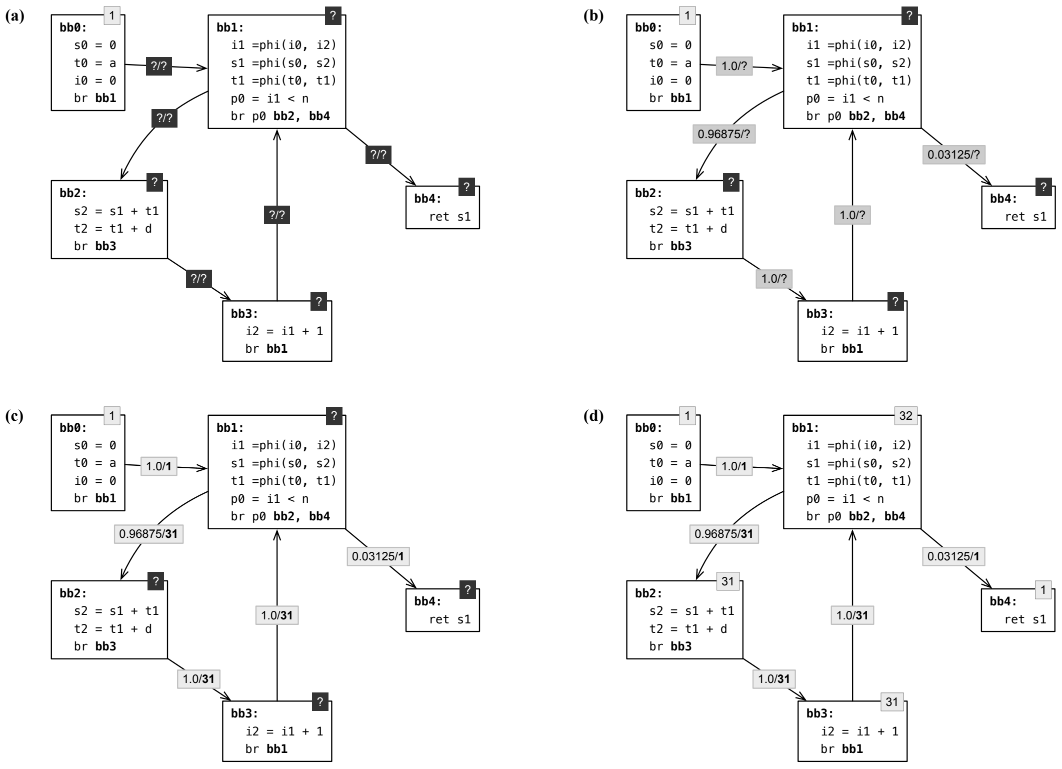
3.3 Prediction with a Generative Model

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.
(2) Parse the labels of basic blocks in each function.
(3) Generate a tailored prompt for each function following the template in Figure 3.



```
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 bbl

bb4:
  ret s1
```

Fig. 2. 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.

(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.

4 Profile Projection Heuristics

This section 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. Besides the control-flow graph G_{opt}, whose hot-order must be inferred, they also take as input a previous version G_{ref} of the program's CFG together with a profile P_e for the edges of G_{ref}. Note that G_{ref} and G_{opt} may differ in both the number of vertices and the structure of edges. All the heuristics in this section, except for the one discussed in Section 4.4, use the same template, which Algorithm 1 shows. The only difference between these heuristics is in the implementation of `block_matching`. The rest of this section discusses the variations that we have evaluated in this paper.

```
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.
```

Fig. 3. 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.

Algorithm 1: Profile projection via block and edge matching. Sections 4.1 and 4.2 discuss different implementations of function `matching_heuristic`. Section 4.3 discusses the implementation of function `block_matching`. The LLM-based projector of Section 4.4 does not use this template, although it solves the same problem.

```
Input: Original program $G_{ref}$ with profile $P_e$, optimized program $G_{opt}$
Output: Projected profile $P''_e$ on $G_{opt}$
Initialize $M_b \leftarrow \emptyset$, $M_e \leftarrow \emptyset$, $P'_e \leftarrow \emptyset$
$M_b \leftarrow$ block_matching($G_{old}, G_{new}, \ldots$ 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
        $M_e[e_{ref}] \leftarrow e_{opt}$
        $P'_e[e_{opt}] \leftarrow P_e[e_{ref}]$
    end
end
Use He et al. [17] to extend $P'_e$ to a total function: $P''_e \leftarrow$ fill_missing_data($P'_e$)
foreach $e_{opt}$ such that $P'_e[e_{opt}] = undef$ do
    $P''_e[e_{opt}] \leftarrow$ missing data
end
Use $P''_e$ and Equation 1 to build $P''_b$
```

4.1 Hash-Based Matching

Two prior techniques, proposed by Wang et al. [40] and Ayupov et al. [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 as a balance between collision resistance and robustness to program transformations. For instance, Ayupov et al. use the following three hash variants to find a match for a basic block b:

Loose: a hash based solely on the set of instruction opcodes in b, ignoring operands.
Strict: a hash based on all instruction opcodes and operands in b, in their exact order of appearance.
Full: a combination of b's strict hash and the loose hashes of its predecessors and successors. This variant is used by Ayupov et al. [2], but not by Wang et al. [40].

To match basic blocks between two versions of a control-flow graph (e.g., G_{ref} and G_{opt} in Definition 2.6), Ayupov et al. apply the hashes in descending order of strictness: first the full hash, then the strict hash, and finally the loose hash if the others fail. Algorithm 2 shows this pattern. Example 4.1 illustrates how this approach works in practice.

Algorithm 2: Hash-based block matching.

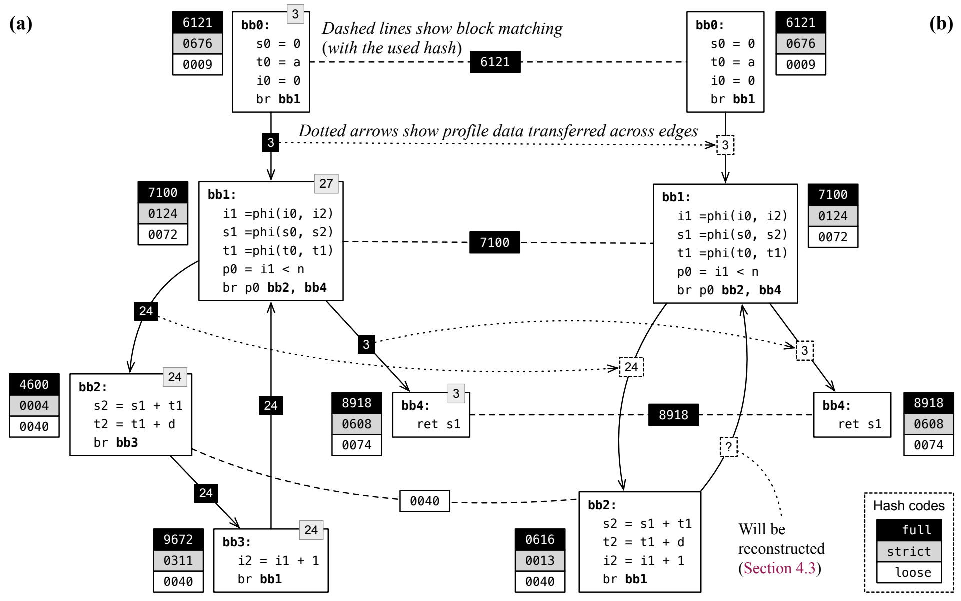
```
Input: Original program $G_{ref}$ with profile $P_e$, optimized program $G_{opt}$
Output: Basic block matching $M_b$
Initialize $M_b \leftarrow \emptyset$
foreach basic block $b_{ref} \in G_{ref}$ do
    $b_{opt} \leftarrow$ match_full($G_{opt}, b_{ref}$) or match_strict($G_{opt}, b_{ref}$) or match_loose($G_{opt}, b_{ref}$)
    if $b_{opt} \neq \emptyset$ then
        $M_b[b_{ref}] \leftarrow b_{opt}$
    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 4.1. Figure 4 (a) shows the hash codes that Ayupov et al. assign to the program from Figure 1 (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) [34]. In LLVM 18.1.8, this optimization merges two blocks from Figure 4 (a), e.g., bb2 and bb3, into a single block bb2 in Figure 4 (b). The full hash successfully matches three blocks between the two versions: bb0, bb1, and bb4. The merged block bb2 in Fig. 4 (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 (a) and bb2 in Figure 4 (b).

As Example 4.1 demonstrates, Ayupov et al.'s method attempts to match blocks using as much contextual information as possible, gradually relaxing the hash definition when a direct match cannot be found. Wang et al.'s technique follows the same principle, but omits the **full** hash.

4.2 HistLoop: Histogram-Based Similarity Matching

This section introduces a heuristic that matches basic blocks by *similarity*. This approach contrasts with those of Wang et al. [40] and Ayupov et al. [2], described in Section 4.1, which rely on exact hash matching. The similarity criterion is defined as follows:


(a)
(b)
Dashed lines show block matching (with the used hash)
Dotted arrows show profile data transferred across edges
Will be reconstructed (Section 4.3)
Hash codes
full
strict
loose
6121
0676
0009
7100
0124
0072
4600
0004
0040
8918
0608
0074
9672
0311
0616
0013
3
27
24
?

```
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
  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
  i2 = i1 + 1
  br bb1

bb4:
  ret s1
```

Fig. 4. (a) The program from Figure 1 (a), annotated with three hash variants for each basic block. (b) The same program after optimization with LLVM's Global Value Numbering.

Definition 4.2 (Opcode Histograms). Let I be a set of instructions ranging over an opcode alphabet L. A *histogram of opcodes*, denoted H_I, is a table that maps each opcode in L to the number of times it occurs in I. If b is a basic block with a set of predecessors $P = \{p_1, p_2, \ldots, p_m\}$ and successors $S = \{s_1, s_2, \ldots, s_n\}$, we define the *Local Histogram* of b as the histogram computed over the set $\{b\} \cup P \cup S$. If $L = \{b_1, b_2, \ldots, b_k\}$ is a *natural loop*[7], we define the *Loop Histogram* as the histogram formed over all basic blocks in L. Finally, given three basic blocks b_1, b_2, and b_3, we say that b_1 is *more similar* to b_2 than to b_3 if the Euclidean distance between the local histograms of b_1 and b_2 is smaller than that between b_1 and b_3.

Histogram-based matching establishes correspondences between the basic blocks of two programs, G_{ref} and G_{opt}, using two algorithms: `loop_matching` and `block_matching`. The former matches loops in the CFG using loop histograms as the similarity criterion. The latter matches blocks in the CFG using local histograms as the similarity criterion. These algorithms replace `block_matching` in Algorithm 1 with the following function composition:
$M_b \leftarrow$ `block_matching`$(G_{ref}, G_{opt},$ `loop_matching`$(G_{ref}, G_{opt}, null, null))$

Algorithm 3, which implements `loop_matching`, recursively matches loops in the program, starting from the least nested to the most nested. The base case occurs when the analyzed loop has size one; that is, it contains a single basic block (not necessarily a loop). Recursion occurs whenever `loop_matching` detects an inner loop, as Example 4.3 explains.

Example 4.3. Figure 5 shows the histogram matching process when projecting a profile from the program in Figure 1 (c) to its optimized version after loop rotation seen in Figure 1 (d). Figure 5 (a) shows the initial call to the `loop_matching` function. The histograms are simplified to display only

[7] A natural loop is a strongly connected component in a control-flow graph with a single entry point. To simplify Algorithm 3, we let l_{ref} and l_{opt} refer to either natural loops or single basic blocks that exist outside any loop, e.g., which have l_{ref}`.size` $= 1$; hence, giving the non-recursive case of Algorithm 3.

Algorithm 3: `loop_matching`
Histogram-Based loop Matching. The function `closest_loop` returns the loop $l_{opt} \in L'_{opt}$ whose histogram has the lowest Euclidean distance to that of l_{ref}. The function `loop_matching` recursively applies this algorithm to nested loops.

```
Input: Reference code $G_{ref}$, optimized code $G_{opt}$, reference loop $L_{ref}$, optimized loop $L_{opt}$
Output: Matching of loops $M_l$ from $L_{ref}$ to $L_{opt}$
Initialize $M_l \leftarrow \emptyset$; $M'_l \leftarrow \emptyset$
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
    $l_{opt} \leftarrow$ closest_loop($L'_{opt}, l_{ref}$)
    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
            $M_l[l_{ref}] \leftarrow l_{opt}$
        else
            $M_l = M_l \cup$ loop_matching($G_{ref}, G_{opt}, l_{ref}, l_{opt}$)
        end
    end
end
```

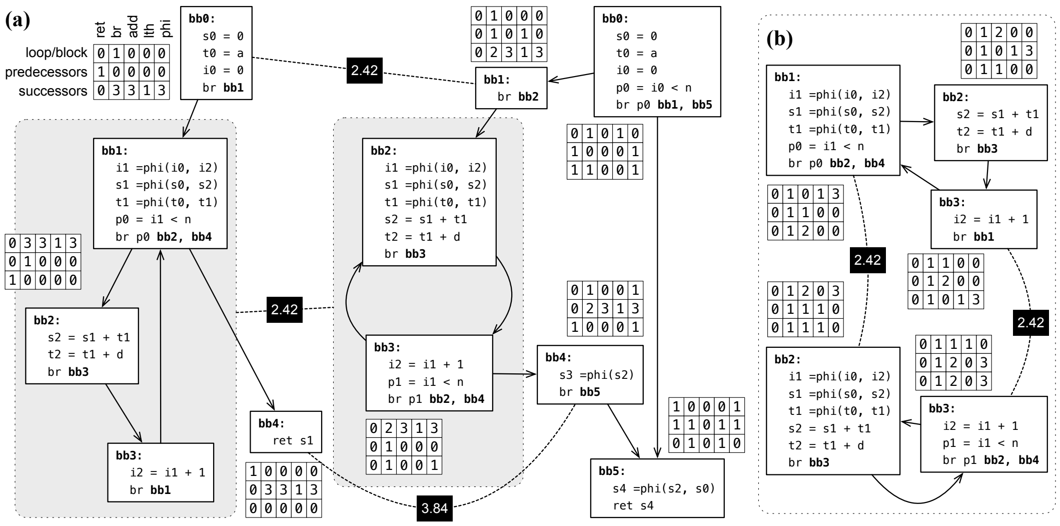
the operands present in the basic blocks: `ret`, `br`, `add`, `lth`, and `phi`. In this phase, a matching is formed between the regions `[bb0]`, `[bb1,bb2,bb3]` and `[bb4]` from the original program and the regions `[bb1]`, `[bb2,bb3]` and `[bb4]` from the optimized program. Figure 5 shows the Euclidean Distance between matched histograms in the matching edges. Subsequently, `loop_matching` is recursively called for the regions with multiple blocks, e.g., `[bb1, bb2, bb3]` (original) and `[bb2, bb3]` (optimized), as shown in Figure 5(b). This results in a matching from `[bb1]` to `[bb2]` and `[bb3]` to `[bb3]`, which Figure 5 (b) shows.

Algorithm 4 refines the block-level matching without overwriting the results from Algorithm 3. In this way, it handles basic blocks that belong either to unmatched loops or to no loop at all. Note

Algorithm 4: `block_matching`
Histogram-Based Block Matching. The function `convert` initializes block-level mappings from loop-level matches by using the headers of matched loops. The function `closest_block` returns the block $b_{opt} \in G_{opt}$ whose histogram has the lowest Euclidean distance to that of b_{ref}.

```
Input: Original program $G_{ref}$, optimized program $G_{opt}$, matching of loops $M_l$
Output: Matching of basic blocks $M_b$ from $G_{ref}$ to $G_{opt}$
Initialize $M_b \leftarrow$ convert($M_l$)
foreach basic block $b_{ref} \in G_{ref}$ do
    $b_{opt} \leftarrow$ closest_block($G_{opt}, b_{ref}$)
    if $b_{opt} \neq \emptyset$ then
        $M_b[b_{ref}] \leftarrow b_{opt}$
    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
```

that HistLoop may still leave some blocks unmatched (i.e., associated with *undef*). This situation arises when G_{opt} contains more basic blocks than G_{ref}. In such cases, only the most similar blocks in G_{opt} are matched, while the remaining ones are left unmatched.



```
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
```

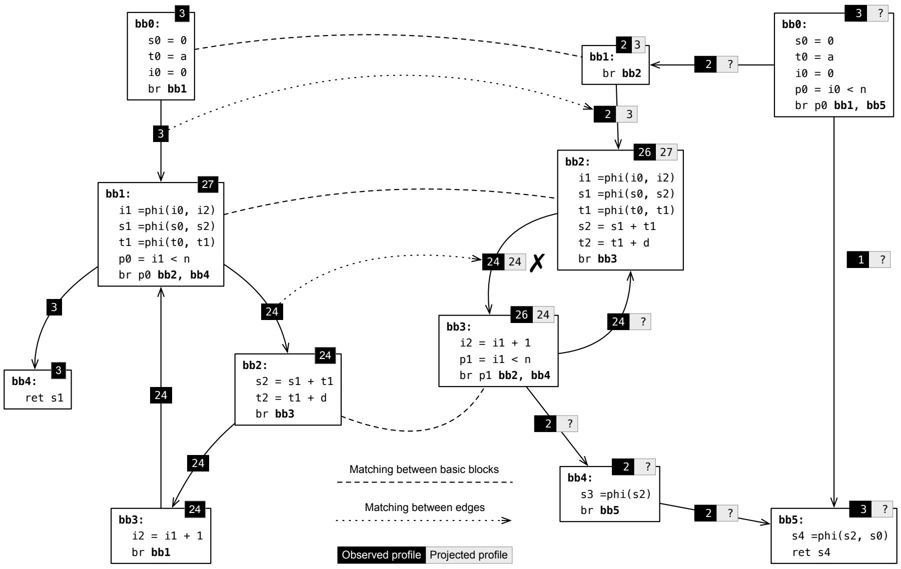
Fig. 5. (a) Matching of block regions using Algorithm 3 (`loop_matching`). We let G_{ref} be the left-side CFG, and G_{opt} the right-side one. (b) Matching of block regions after a recursive call of Algorithm 3.

Example 4.4. After `loop_matching` executes, two blocks in the optimized program of Figure 5 (a) remain unmatched: `bb0` and `bb5`. Similarly, one block from the original program has no match: `bb2`. Algorithm 4 then pairs `bb2` from G_{ref} with `bb5` in G_{opt}, since this pair has a smaller Euclidean distance than the pair (`bb2`, `bb0`). Note that this match between a block inside a loop and a block outside any loop may introduce small inaccuracies in the projected profile. However, such discrepancies are later corrected by the algorithm described in Section 4.3.

4.3 Filling Out Missing Profile Information

After matching the basic blocks and edges of G_{ref} and G_{opt}, and assigning the corresponding weights from P_e'', Algorithm 1 must fill in any missing profile information. We adopt the same method as Ayupov et al. [2], who build on the work of He et al. [17]. In their approach, the inference of missing profile data is reduced to an instance of the Minimum-Cost Flow (MCF) problem [10]. If a complete profile cannot be inferred, He et al. [17] distribute the remaining flow according to a branch-probability heuristic. Following Ayupov et al. [2], we use a uniform heuristic that assigns equal probability to all outgoing edges. Example 4.5 illustrates how this inference works.

Example 4.5. Figure 6 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).


```
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

bb4:
  ret s1

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

bb3:
  i2 = i1 + 1
  br bb1
```

```
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
```

Fig. 6. Matching between the control-flow graphs of Figure 1 (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.

Example 4.5 illustrates that direct projection of profile data can violate the law of conservation of flow. For instance, in Figure 6, the edge from `bb2` to `bb3` in G_{opt} is the only outgoing edge of `bb2`, yet its edge frequency does not match the block frequency; hence, breaking flow conservation. The reconstruction procedure of He et al. [17] resolves such inconsistencies, as shown next.

Example 4.6. Figure 7 shows how He et al. [17]'s algorithm works on the left-side program seen in Figure 6. 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, He et al.'s algorithm redistributes the missing flow proportionally according to branch probabilities.

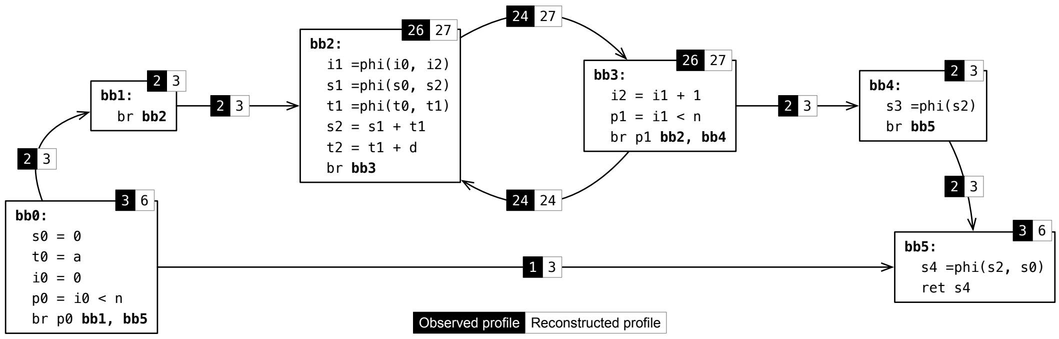


Fig. 7. Result of applying He et al. [17]'s reconstruction algorithm to the optimized program in Figure 6. The observed and reconstructed profiles are shown side by side, with adjusted frequencies ensuring flow conservation across the CFG.

4.4 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 8. 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.
- 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.

```
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.
```

Fig. 8. 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.

- **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.
- **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.

5 Experimental Evaluation

This section investigates several research questions concerning the heuristics introduced in Sections 3 and 4. For clarity, we group these heuristics into two categories: *LLM-based*, comprising the techniques described in Sections 3.3 and 4.4, and *classic*, comprising all the others. 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?

Evaluation Metric. We analyze the destructive effects that LLVM optimizations such as constant propagation 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 Game of Definition 2.3 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 instrumented program. Following Definition 2.3, our goal is to minimize the 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:

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

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

Benchmarks. We evaluate our approach using the cBench benchmark suite [13], 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`, even in the absence of instrumentation. 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 4.4 did not yield valid results for every instance of the Hot-Order Game. 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. Our approach was implemented in LLVM 18.1.8. For instrumentation, we used the Nisse framework [12], 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* and *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. 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.

Discussion. Table 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, matching the theoretical result in Theorem 3.1. 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.

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 1. Relative accuracy (Def. 5.1) of classic prediction heuristics.

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 with respect to the Profile Projection Problem introduced in Definition 2.6. 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 2 summarizes our observations. In general, projection tends to outperform prediction, as shown by comparing Tables 1 and 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	**Ayupov** (Sec. 4.1)	89.90%	65.28%	64.71%	63.95%
	HistLoop (Sec. 4.2)	**95.55%**	**74.19%**	**73.63%**	**72.52%**
-O1	**Ayupov**	64.51%	91.60%	83.29%	81.57%
	HistLoop	**71.01%**	**97.17%**	**89.36%**	**87.61%**
-O2	**Ayupov**	64.44%	83.55%	91.78%	89.55%
	HistLoop	**71.73%**	**90.14%**	**96.95%**	**94.96%**
-O3	**Ayupov**	63.64%	81.34%	89.05%	91.19%
	HistLoop	**70.51%**	**88.02%**	**94.76%**	**96.85%**

Table 2. Relative accuracy (Def. 5.1) of classic profile projection heuristics.

The histogram-based matching heuristic of Section 4.2 consistently outperforms the approach proposed by Ayupov et al. [2] 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 Ayupov et al.'s technique lies in its limited robustness to structural transformations such as loop rotation (illustrated in Figure 1) 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, as in Ayupov et al.'s method.

5.3 RQ3 – Comparison with LLM-Based Profile Prediction

This section compares the LLM-based profile predictor described in Section 3.3 against the classic heuristics presented in Sections 3.1 and 3.2. 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. As previously discussed, the number of functions had to be reduced because the generative model occasionally failed to produce valid orderings. Invalid sequences either contained fewer or more basic blocks than the original control-flow graph, leading to an inconsistent block mapping.

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 3. Comparison between classic prediction heuristics and the LLM-based predictor.

Discussion. Table 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.

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 described in Section 4.4 with the classic heuristics presented in Section 4. 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. The universe of functions is smaller than that of Section 5.3, since the prompts required for projection are substantially larger. These prompts include two control-flow graphs, as illustrated in Figure 8. The larger the prompt, the higher the likelihood that the generative model produces malformed or incomplete outputs.

Discussion. Table 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 4.1 and 4.2. 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.

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

Table 4. Comparison between classic projection heuristics and LLM-based projection.

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.

5.5 RQ5 – Impact of Different Optimizations on Profile Projection

To understand why our method 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 our profile projection heuristics.

LLVM includes hundreds of optimization passes, making exhaustive analysis impractical. Therefore, we restrict our study to the 35 optimizations identified by Silva et al. [37] as the most common in effective `clang` optimization sequences. Unlike previous sections, this experiment uses a single input to isolate the effects of optimizations on the quality of the projected profile. The adopted methodology is as follows:

(1) Profile 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 Ayupov et al. [2]'s heuristic or the histogram-based heuristic from Section 4.2 to reconstruct the profile of G_{opt}, obtaining the projected ordering H_p;
(4) Profile 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.

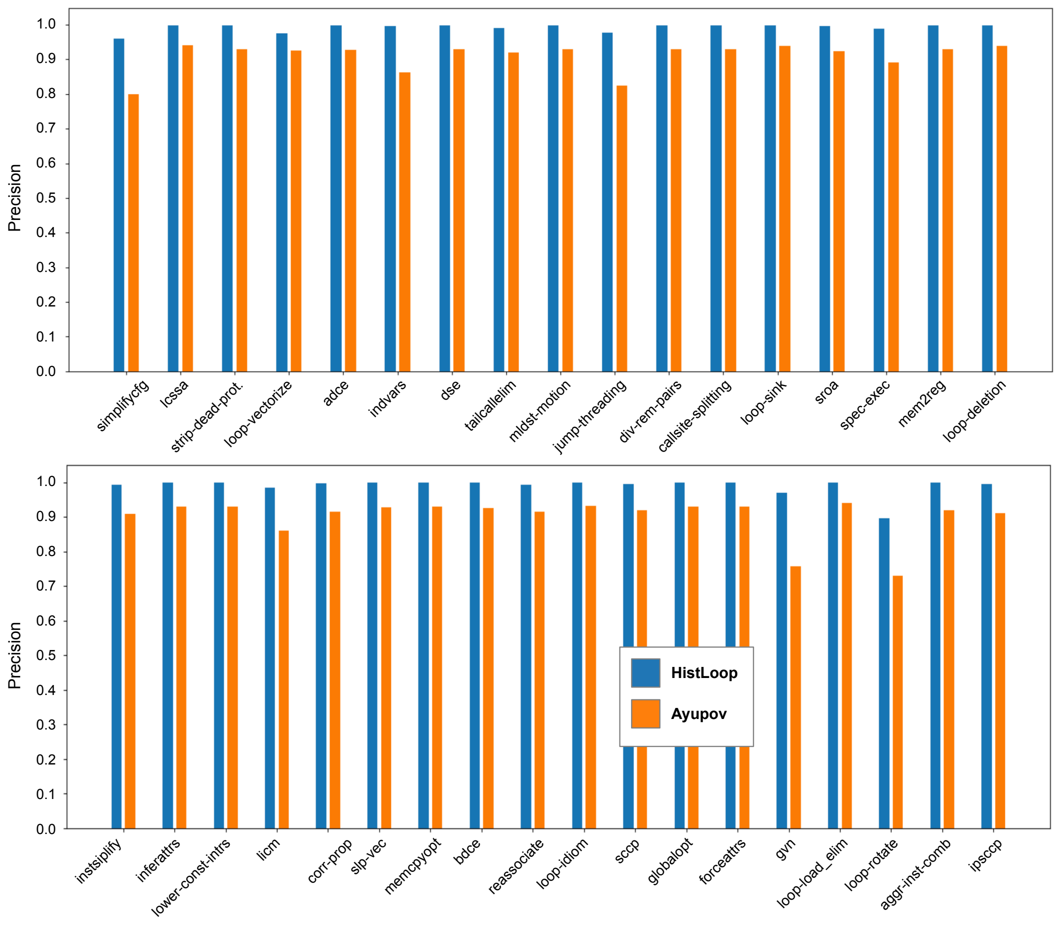


Fig. 9. Impact of different optimizations on the accuracy of classic profile projection heuristics. Blue bars show accuracy for HistLoop, while orange bars show accuracy for Ayupov et al.'s technique.

Discussion. Figure 9 presents the accuracy results for the HistLoop heuristic (blue bars) and Ayupov et al.'s heuristic (orange bars). We observe that most optimizations have minimal impact on HistLoop, whereas Ayupov et al.'s approach is more sensitive to individual transformations. However, two optimization passes significantly reduce the accuracy of our histogram-based matching algorithm: `simplifycfg` and `loop-rotate`.

The `simplifycfg` pass alone does not drastically harm projection accuracy; however, it is one of the most frequently applied transformations in the LLVM pipeline [37]. 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. A promising direction for future work would be to determine how often profile information must be regenerated or revalidated during the optimization process.

In contrast, `loop-rotate` poses a more intrinsic challenge to our method. This pass modifies the internal control-flow structure of loops, often changing entry and exit points in ways that alter execution frequencies. Although the loop-matching phase described in Section 4.2 is designed to accommodate structural changes, it cannot infer accurate iteration frequencies when such information is absent from the reference program. Consequently, loop rotation remains a fundamental source of inaccuracy for structure-based profile projection.

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.

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.

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. Execution time of classic profile prediction heuristics.

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

Table 6. Execution time of classic profile projection heuristics.

Discussion. Table 5 reports the running times of the profile prediction heuristics, while Table 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 approach of Ayupov et al. [2], and two orders of magnitude faster than the histogram-based heuristic. 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 method of Ayupov et al. [2] outperforms the histogram-based approach (Section 4.2) because it matches basic blocks in linear time using hash lookups, whereas the histogram-based method performs pairwise similarity comparisons across blocks or loops—a computationally more expensive process. Nevertheless, the histogram-based heuristic remains practical, consistently completing in under one minute across all 168 functions, regardless of the optimization level.

6 Related Work

This paper addresses the problem of projecting profile information from one version to another version of this function. This task needs the resolution of a problem called intra-procedural binary correspondence: given two control-flow graphs, the maximal correspondence between them is searched. The intra-procedural correspondence is used in malware identification, redundancy elimination, and plagiarism detection. Two recent revisions [1, 16] mention only one work on inter-procedural correspondence as a way to do profile projection: the BMAT approach from Wang et al. [40]. The same revision shows that the inter-procedural version of binary correspondence is more commonly discussed in the profile reuse context. This is also the conclusion from the revision of Kim and Notkin [23], which focus on the binary correspondence between versions of the same program. Either the intra- and inter-procedural binary correspondence frames on a more general framework the literature calls the Binary Diffing problems.

Binary Diffing. This term is widely used in the programming language literature, with different meanings for different authors. For instance, Hemel et al. [18] uses this expression to refer to the problem of determining whether two binaries are from the same source code or different versions of the same source code. Adopting a slightly different understanding of the term, Ming et al. [26] want to determine if two binaries implement similar semantics, i.e., they solve the same problem. Kim et al. [22], in turn, understands diffing as intra-procedural code correspondence. These divergent interpretations give two distinct approaches to the problem known as binary diffing [1, 16]. On the one hand, we have stochastic techniques – machine learning models, like the one from Shin et al. [36] – to solve the semantic problem of determining when two binaries execute similar actions. On the other hand, we have hash-based approaches to align binary code, either finding correspondence between functions (inter-procedural correspondence) or between control-flow graphs (intra-procedural correspondence). The profile projection techniques seen in section 4 could be used to solve the latter version of the problem.

Inter-Procedural Correspondence. In the words of Flake [11], 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*". Typically, inter-procedural correspondence extracts characteristics from the many functions that compose the program and tries to combine them based on these characteristics. Such characteristics can be structural, for instance, properties from the control-flow graph of the function [25], such as the number of nodes and edges, circumference, tree-width, etc. They can also be semantical, like instructions opcodes histograms [8] and many vectors built as combinations of instructions from the program [9]. The main motivation for the development of inter-procedural techniques is the need to deal with binary code without high-level information, like function names. Notice that, in the context of this work, such motivation does not apply, since quality industrial systems that do profile-guided optimization in big binaries have access to the function names [6, 30, 32, 39].

Intra-Procedural Correspondence. The branching mapping problem discussed in section 2 is an instance of intra-procedural binary correspondence. Many works deal with this problem, as discussed in the literature review by Kim and Notkin [23]. In this case, the challenge is to find a correspondence between anchor points of two functions. The anchor point type defines the solution nature. There are three main anchor point types: nodes on abstract syntax trees [42] (when there is access to the source code); instructions on linear sequences of binary code [19]; or vertex in control-flow graphs. The profile projection techniques seen in section 4 fit in the third category. Common solutions for graph correspondence are based on graph isomorphism or sequence alignment, In the first case, the graphs can be either abstract syntax trees or 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, 21].

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 re-running 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.

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. Thus, we plan to explore richer ways to support LLM-based profiling. In particular, we aim 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.

Data Availability Statement

The work is publicly available on https://github.com/lac-dcc/hydra, under the Apache-3.0 license. Using the LLM-based heuristics requires paid access to the GPT-4o API. If the paper is accepted for publications, the authors shall submit an artifact.

Acknowledgment

This project was supported by FAPEMIG (Grant APQ-00440-23), CNPq (Grant #444127/2024-0), CAPES (PrInt), and Google, which sponsored Elisa Fröhlich's scholarship. We are grateful to Xinliang (David) Li and Victor Lee for their efforts in making the Google sponsorship possible.

References

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

[2] Amir Ayupov, Maksim Panchenko, and Sergey Pupyrev. 2024. Stale Profile Matching. In *Compiler Construction* (Edinburgh, United Kingdom). Association for Computing Machinery, New York, NY, USA, 162–173. doi:10.1145/3640537.3641573

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

[4] Thomas Ball and James R. Larus. 1993. Branch Prediction for Free. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 300–313. doi:10.1145/155090.155119

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

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

[7] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. 1986. Code motion of control structures in high-level languages. In *POPL* (St. Petersburg Beach, Florida). Association for Computing Machinery, New York, NY, USA, 70–85. doi:10.1145/512644.512651

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

[9] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *SP*. IEEE, Washington, DC, USA, 472–489. doi:10.1109/SP.2019.00003

[10] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (April 1972), 248–264. doi:10.1145/321694.321699

[11] Halvar Flake. 2004. Structural Comparison of Executable Objects. In *DIMVA (LNI, Vol. P-46)*, Ulrich Flegel and Michael Meier (Eds.). GI, Dortmund, Germany, 161–173. https://dl.gi.de/20.500.12116/29199

[12] Leon Frenot and Fernando Magno Quintão Pereira. 2024. Reducing the Overhead of Exact Profiling by Reusing Affine Variables. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 150–161. doi:10.1145/3640537.3641569

[13] Grigori Fursin and Olivier Temam. 2011. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.* 7, 4, Article 20 (Dec. 2011), 29 pages. doi:10.1145/1880043.1880047

[14] T. Glek and J. Hubicka. 2010. Optimizing real world applications with GCC Link Time Optimization. arXiv:1010.2196 [cs.PL] https://arxiv.org/abs/1010.2196

[15] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. *SIGPLAN Not.* 17, 6 (jun 1982), 120–126. doi:10.1145/872726.806987

[16] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (apr 2021), 38 pages. doi:10.1145/3446371

[17] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile Inference Revisited. *Proc. ACM Program. Lang.* 6, POPL, Article 52 (jan 2022), 24 pages. doi:10.1145/3498714

[18] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding Software License Violations through Binary Code Clone Detection. In *MSR*. Association for Computing Machinery, New York, NY, USA, 63–72. doi:10.1145/1985441.1985453

[19] He Huang, Amr M. Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *ASIA CCS*. Association for Computing Machinery, New York, NY, USA, 155–166. doi:10.1145/3052973.3052974

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

[21] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A Search Engine for Binary Code. In *MSR* (San Francisco, CA, USA). IEEE Press, New York, NY, USA, 329–338.

[22] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving Cross-Platform Binary Analysis Using Representation Learning via Graph Alignment. In *ISSTA* (Virtual, South Korea). Association for Computing Machinery, New York, NY, USA, 151–163. doi:10.1145/3533767.3534383

[23] Miryung Kim and David Notkin. 2006. Program Element Matching for Multi-Version Program Analyses. In *MSR* (Shanghai, China). Association for Computing Machinery, New York, NY, USA, 58–64. doi:10.1145/1137983.1137999

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

[25] Jiang Ming, Meng Pan, and Debin Gao. 2012. IBinHunt: Binary Hunting with Inter-Procedural Control Flow. In *ICISC ('12)*. Springer-Verlag, Berlin, Heidelberg, 92–109. doi:10.1007/978-3-642-37682-5_8

[26] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-Based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *SEC* (Vancouver, BC, Canada). USENIX Association, USA, 253–270.

[27] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Pereira. 2023. Beetle: A Feature-Based Approach to Reduce Staleness in Profile Data. doi:10.22541/au.168898868.83064146/v1

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

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

[30] Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 151–165.

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

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

[33] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (July 2010), 65–79. doi:10.1109/MM.2010.68

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

[35] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *ASPLOS* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 617–631. doi:10.1145/3575693.3575727

[36] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *SEC* (Washington, D.C.). USENIX Association, USA, 611–626.

[37] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. 2021. 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* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/3446804.3446849

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

[39] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. 2022. One Profile Fits All: Profile-Guided Linux Kernel Optimizations for Data Center Applications. *SIGOPS Oper. Syst. Rev.* 56, 1 (jun 2022), 26–33. doi:10.1145/3544497.3544502

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

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

[42] Wuu Yang. 1991. Identifying Syntactic Differences between Two Programs. *Softw. Pract. Exper.* 21, 7 (jun 1991), 739–755. doi:10.1002/spe.4380210706