Profile Inference Revisited

WENLEI HE, Facebook Inc., USA
JULIÁN MESTRE, Facebook Inc., USA and University of Sydney, Australia
SERGEY PUPYREV, Facebook Inc., USA
LEI WANG, Facebook Inc., USA
HONGTAO YU, Facebook Inc., USA

Profile-guided optimization (PGO) is an important component in modern compilers. By allowing the compiler to leverage the program's dynamic behavior, it can often generate substantially faster binaries. Sampling-based profiling is the state-of-the-art technique for collecting execution profiles in data-center environments. However, the lowered profile accuracy caused by sampling fully optimized binary often hurts the benefits of PGO; thus, an important problem is to overcome the inaccuracy in a profile after it is collected. In this paper we tackle the problem, which is also known as *profile inference* and *profile rectification*.

We investigate the classical approach for profile inference, based on computing minimum-cost maximum flows in a control-flow graph, and develop an extended model capturing the desired properties of real-world profiles. Next we provide a solid theoretical foundation of the corresponding optimization problem by studying its algorithmic aspects. We then describe a new efficient algorithm for the problem along with its implementation in an open-source compiler. An extensive evaluation of the algorithm and existing profile inference techniques on a variety of applications, including Facebook production workloads and SPEC CPU benchmarks, indicates that the new method outperforms its competitors by significantly improving the accuracy of profile data and the performance of generated binaries.

CCS Concepts: • **Software and its engineering** → *Compilers*; • **Theory of computation** → **Graph algorithms analysis**.

Additional Key Words and Phrases: compilers, profile-guided optimizations, network flows

1 INTRODUCTION

In a connected world with ubiquitous AI and cloud applications that are underpinned by massive back-end servers, optimizing server efficiency is becoming increasingly popular and demanding. Profile-guided optimization (PGO) is an important step in modern compilers for improving performance of large-scale applications based on their run-time behavior. The technique, also known as feedback-driven optimization (FDO), leverages program execution profiles, such as the execution frequencies of basic blocks and function invocations, to guide compilers to optimize critical paths of a program more selectively and effectively. Nowadays PGO is an essential part of most commercial and open-source compilers for static languages; it is also used for dynamic languages, as a part of Just-In-Time (JIT) compilation. PGO has been widely adopted to speed up warehouse applications in industry, providing double-digit percentage performance boost.

Traditionally, profile-guided optimizations are implemented by means of compile-time instrumentation which injects code that counts the execution frequencies of basic blocks, jumps, and function calls into the application. This approach, however, incurs significant operational overhead, as it requires extra training build and profiling run. Furthermore, the code injected by the compiler introduces enormous performance overhead (up to 10x slowdown in large-scale binaries), which

Authors' addresses:
Wenlei He, wenlei@fb.com, Facebook Inc., Menlo Park, CA, USA;
Julián Mestre, julianmestre@fb.com, Facebook Inc., Menlo Park, CA, USA, julian.mestre@sydney.edu.au, University of Sydney, Australia;
Sergey Pupyrev, spupyrev@fb.com, Facebook Inc., Menlo Park, CA, USA;
Lei Wang, wlei@fb.com, Facebook Inc., Menlo Park, CA, USA;
Hongtao Yu, hoy@fb.com, Facebook Inc., Menlo Park, CA, USA..

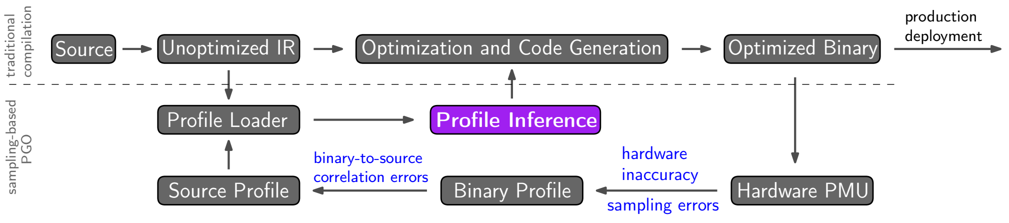


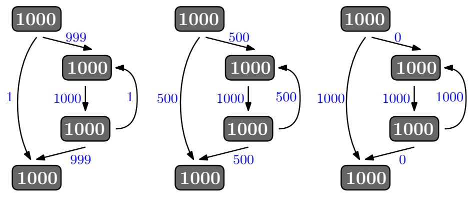
Fig. 1. Life cycle of profile data in sampling-based PGO: the complex pipeline introduces errors and inconsistencies at various stages degrading profile quality, and profile inference is needed before the profile is fed back into the compiler.

may alter the program's default behavior and make the collected profiles non-representative. As a result, a new methodology, namely sampling-based PGO, has emerged with the help of hardware performance monitoring units (PMU) available on modern processors. Sampling-based PGO enables profiling in the production environment with a negligible runtime overhead, while avoiding extra training builds. LLVM's AutoFDO [Chen et al. 2016] and BOLT [Panchenko et al. 2019] are examples of widely utilized frameworks for optimizing data-center workloads based on the technology.

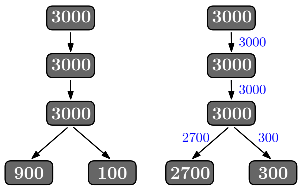
While sampling-based PGO lowers the entry barrier for PGO, it is not delivering the same level of performance boost compared to the instrumentation-based counterpart. Since the compiler applies the same optimizations with sampling-based and instrumentation-based profiles, the performance difference is determined by the quality of profile data. There are two main sources of inaccuracies:

- To prevent PMUs from slowing down the execution, modern processors come with several trade-offs that make it impossible to exactly attribute a hardware event to the corresponding instruction address [Chen et al. 2013; Xu et al. 2019; Yi et al. 2020]. As a result, the collected execution frequencies of some basic blocks are biased or mis-attributed, especially for small-sized functions with heavyweight instructions.
- Sampling-based profilers associate execution counts to instruction addresses, while a compiler needs to associate the counts to basic blocks in the program. The problem is illustrated in Figure 1. A typical life cycle of a sampling-based profile starts from the hardware PMU sampling; it then goes through the initial post-processing and aggregation that result in a binary-level profile. Later on, the binary-level profile is converted to the source-level profile that can be consumed by the compiler. Finally, the profile is adjusted throughout the compiler optimization pipeline to guide various optimizations. There is loss of profile fidelity along each of the steps.

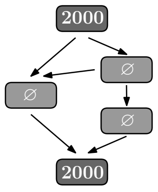
For binary optimizers like BOLT, hardware inaccuracy is the main source of profile quality degradation. In contrast, for compiler PGO, the quality loss is caused by profile correlation from the fully optimized binary back to the source (non-optimized) internal representation (IR). Therefore, an important problem for sampling-based PGO is to fix the errors in a profile after it is collected. This problem, also known as *profile inference* and *profile rectification*, is the main topic of the paper. The primary goal of the work is to overcome profile quality loss during a complex compilation pipeline of sampling-based PGO. Each step of the profile life cycle poses unique challenges for profile inference, and it is highly non-trivial to handle them in a single inference component. Next section overviews the main challenges.



(a) Multiple possible edge frequencies corresponding to the same sample block counts



(b) Preserving relative sample block counts is desirable



(c) Dangling blocks with unknown sample counts

Fig. 2. Primary challenges in profile inference

1.1 Main Challenges

Here we identify three aspects of the inference problem making it complicated.

- As explained earlier, the profiles used for sampling-based PGO are collected through sampling, as collecting full instrumentation-based profiles incurs unacceptable overhead. This is especially true for JIT-based compilers, where the profiles are collected on-the-fly and the extra overhead directly translates to performance regressions. Sampling unavoidably introduces statistical inconsistencies in the profile data, which impairs the benefits of PGO. For example, the sample count of a basic block is not always equal to the sum of the counts of its successors or predecessors. Another common scenario is when some relatively cold blocks are not receiving any samples, although such blocks are known to be executed as they dominate basic blocks with samples.
  Earlier works observed hardware-related problems that lead to inaccurate results, and suggested several tricks that help to correct sample counts. Chen et al. [2013] utilize *Precise Event Based Sampling* available on modern Intel x86 processors and propose sampling multiple hardware events to mitigate these hardware bias effects. A later study by Wu et al. [2013] evaluates an approach of varying the sampling rate but finds it does not significantly improve the accuracy of collected counts. More recent works of Nowak et al. [2015], Chen et al. [2016], and Panchenko et al. [2019] adopt *Last Branch Records* (LBRs) technology to count basic block execution frequency. However, even LBRs cannot eliminate all design flaws in modern processors, and raw collected frequencies are still biased [Xu et al. 2019; Yi et al. 2020]. Therefore, a post-processing technique to rectify the measurement inaccuracies is needed.
- Typically sampling-based profile data consists of frequencies of individual basic blocks. However, many PGO optimization passes, such as basic block reordering or hot-cold function splitting, heavily rely on edge frequency information. Thus, the derivation of edge frequencies from the basic block sample counts is a core component of the sample profile support. However, determining edge frequencies from the block counts is not always possible: a given block frequency annotation may correspond to multiple edge frequency annotations [Chen et al. 2016; Levin et al. 2008]. Figure 2a illustrates such an instance; notice that different edge frequency assignments may yield different layouts of the basic blocks in the function, which might greatly affect the performance of the generated binary [Newell and Pupyrev 2020].
  One approach for solving the problem was suggested by Probert [1982] and later studied by Ball and Larus [1994], who show how to optimally insert monitoring code for collecting edge frequency profiles. Their technique breaks all *critical* edges in the control-flow graph and inserts counters, which makes it possible to reconstruct the complete edge frequency profile, while minimizing extra overhead. As mentioned earlier, such an instrumentation-based approach is often unacceptable in modern data-center environments. An arguably more practical approach

```
double flow_cost(network_t *net) {
  ...
  for ( i = 0; i < net->m ; i++ ) {
    if ( arc[i].ident == AT_UPPER )
      arc[i].flow = (flow_t)1;
    else
      arc[i].flow = (flow_t)0;
  }
  ...
}
```

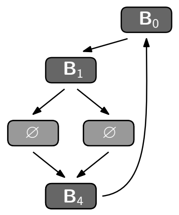
(a) source code

```
<flow_cost>:
  ...
  cmp R1,10
  br L1, L2
L1:
  mov 1, R0
  br L3
L2:
  mov 0, R0
  br L3
L3:
  ...
```

(b) pseudo-IR before optimization

```
<flow_cost>:
  ...
  mov 0, R0
  cmp R1, 10
  // conditional addition
  addeq 1, R0
  ...
```

(c) pseudo-IR after optimization



(d) corresponding control-flow graph

Fig. 3. An instance with *dangling* basic blocks as a result of the *if-conversion* compiler optimization: profile samples are collected on an optimized binary (c), while inference is applied on an non-optimized one (b).

is initiated by Wu and Larus [1994] who design heuristics for predicting the edge frequency profiles. While predicting branch probabilities perfectly can completely eliminate the need of profile collection, we stress that even the best-known techniques based on modern machine learning achieve relatively low profile accuracy [Calder et al. 1997; Shih et al. 2021].
Estimating edge frequencies by coupling block sample counts with statically predicted branch probabilities is a promising direction that we explore in the paper. For example, the realistic profile for the instance in Figure 2a can be inferred by observing that loop back edges typically have a very high probability to be taken.

- Finally, an important challenge is caused by the fact that the sample profile is collected on a fully optimized binary, while the block and edge frequencies are consumed on an early stage of the compilation that operates with a non-optimized IR [Chen et al. 2016; Novillo 2014]. As a result, some of the basic blocks may not have associated sample counts. We call such blocks *dangling*. To illustrate the problem, refer to Figure 3 in which two basic blocks are not present in the optimized binary (and hence, receive no samples during profiling). Yet an inference is applied for a non-optimized binary and an algorithm needs to deduce counts for the two blocks.
While the number of dangling blocks varies for different binaries and functions, we have encountered instances where up to 50% of the blocks are reported as dangling.

1.2 Our Contributions

The primary contributions of this work are summarized below.

- We thoroughly investigate and identify opportunities for improvements in the classical approach for profile inference, initiated by Levin, Newman, and Haber [2008]. Then we extend the model and suggest a new optimization problem with the objectives and constraints capturing the desired properties of profile data. We emphasize that some of the constraints were overlooked by the community yet they are essential in the context of profile inference.
- Next we study theoretical aspects of the new problem. We show that it is computationally APX-hard in general, that is, there exists a constant $\alpha > 1$ such that the problem does not admit an α-approximation unless P = NP, and design a new algorithm that nevertheless works well for the instances we commonly encounter in real-world benchmarks. We develop a noise model for block sample counts and an efficient algorithm for computing a maximum likelihood estimator of the underlying ground truth. Finally, we give an algorithm that can recover vertex weights from edge weights and use it to deal with incomplete data such as dangling blocks.
- We then show that our theoretical framework yields an algorithm that scales to real-world workloads without significant impact on the running time of a compiler or a PGO tool. We

describe the details of our implementation and show that it can be engineered to work effectively at scale. To this end, we amplify the existing approach for computing the minimum-cost maximum flow on a graph by new powerful yet simple heuristics that speed it up by $12x$.
- Finally, we extensively evaluate the new algorithm on a variety of applications, including Facebook production workloads, open-source compilers, Clang and GCC, and SPEC CPU 2017 benchmarks. The experiments indicate that the new method significantly outperforms the state-of-the-art technique, achieving an average of 93.85% overlap with the ground-truth profile. Performance evaluation of the optimized binaries generated with the new profile shows an average increase of PGO speedup by up to 3.5% on large data-center workloads. We have open sourced the code of our new algorithm as a part of LLVM [Lattner and Adve 2004].

The paper is organized as follows. We first discuss existing approaches for profile inference and their limitations in Section 2.1. Then in Section 2.2 a new formal model is suggested, while Section 3 presents a detailed theoretical analysis and algorithms for the corresponding optimization problem. Based on the theoretical findings, we develop a new method for profile inference and discuss some implementation details in Section 4. Next, in Section 5, we present experimental results, that are followed by a discussion of related works in Section 6. We conclude the paper and propose possible future directions in Section 7.

2 THE PROFILE INFERENCE PROBLEM

In this section we investigate existing approaches for profile inference, identify their limitations, and suggest a new formal optimization model for the problem.

2.1 Earlier Approaches and Their Limitations

Much work has been done to alleviate profile inaccuracies. Levin, Newman, and Haber [2008] propose an approach based on the Minimum Cost Flow (MCF) problem that is considered the state-of-the-art. The technique has been extended and adopted by multiple compilers and PGO systems [Chen et al. 2013; Panchenko et al. 2019; Ramasamy et al. 2008]. A variant of the method that iteratively propagates block frequencies across the edges of the control-flow graph is also widely utilized [Chen et al. 2016; Novillo 2014; Ottoni 2018]. Next we discuss limitations of the approaches.

- First, the core component of a classical flow-based algorithm is finding the minimum-cost maximum flow on an auxiliary network constructed from the input control-flow graph. Such a network is composed of nodes and edges with fixed capacities and costs. The edge capacities and costs need to be carefully chosen so that the algorithm yields realistic edge frequencies. While there is freedom in assigning edge costs, Figure 2b illustrates that the computed edge frequencies can be inaccurate for any cost function. Here the two leaf blocks are under-sampled and one needs to distribute extra counts among the two leaves. Arguably the correct inference is to split the excess in proportion of $900 : 100$ so that the observed ratio of the execution counts is preserved. However, a classical MCF algorithm either sends all the flow to the left leaf (thus, making the counts $2900 : 100$) or to the right one (thus, making the counts $900 : 2100$); the assignment depends on the relative costs of the two leaf edges in the auxiliary network.
- Second, classical MCF algorithms cannot guarantee that the resulting edge frequencies correspond to a *connected* flow in the control-flow graph. For an instance in Figure 2a, an MCF-based inference may produce an output in which the hot loop is disconnected from the entry block (refer to the rightmost instance in Figure 2a). Furthermore, creating a connected minimum-cost maximum flow is a computationally hard problem, as we discuss in Section 3. Somewhat surprisingly, this issue has been overlooked in prior works, yet it can result in significant performance regressions of the compiled binaries.

- Finally, finding a minimum-cost maximum flow in a graph is computationally an expensive operation. Indeed, the best strongly-polynomial time algorithm known for this problem has worst-case complexity $O\big(|E|\log|V|(|E|+|V|\log|V|)\big)$ [Orlin 1988], where $|V|$ and $|E|$ are the numbers of vertices and edges in the control-flow graph, respectively. While the performance of MCF can be tolerable for small and medium-sized binaries, the overhead of the approach for building large-scale data-center applications might be unacceptable. We emphasize that the poor performance of general purpose MCF solvers is one of the reasons that led other authors to seek for an alternative solution [Chen et al. 2016; Novillo 2014].

2.2 Optimization Model

We model the task of profile inference as the following optimization problem. The input is a directed (possibly cyclic) control-flow graph for a binary function, denoted $G = (V, E)$. The vertices of G correspond to basic blocks and directed edges represent jumps between the blocks. We assume that the graph contains a unique source, $s^* \in V$ (that is, the entry block of the function) but may have multiple sinks (exit blocks), denoted $T^* \subset V$, that are reachable from s^* via a directed path.

Also a part of the input is a vertex weight $w(v) \geq 0$ for $v \in V$ that represents the execution count of the block as reported by the profile data. As noticed above, the counts are imprecise and serve just as an estimation of real block execution counts. Furthermore, some of the blocks are missing from the profile, which is indicated by $w(v) = \varnothing$; we call such vertices *dangling*. For every branch of the binary function, the input specifies estimated probabilities of taking corresponding edges. Formally, for every non-exit vertex $v \in V$ with outgoing edges $(v, u_1), \dots, (v, u_d)$ for some $d \geq 1$, edge weight $0 \leq w(v, u_i) \leq 1$ is the probability of jumping from block v to its successor u_i for $1 \leq i \leq d$. The probabilities are assumed to be consistent, that is, $\sum_{i=1}^{d} w(v, u_i) = 1$. Typically, we have $d = 1$ (which corresponds to fall-through or unconditional jumps) or $d = 2$ (conditional branches such as `if/else` constructs). However, the out-degree of a vertex can be significantly larger, for example, in `switch` statements or indirect branches.

The high-level goal of profile inference is to reconstruct the most "realistic" behavior of the binary function. Specifically, we focus on two outcomes: execution counts for all $v \in V$ and all $e \in E$. Next we formalize the desired constraints and objectives.

Constraints. A real execution of a binary function starts at the entry block, then traverses the blocks in some order along the edges, and ends at an exit block. Thus, the computed vertex and edge counts have to satisfy so-called *flow conservation* rules. Denote $f(v) \geq 0, v \in V$ and $f(u, v) \geq 0, (u, v) \in E$ to be vertex and edge counts, respectively. Then

$$f(v) = \sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w) \qquad \text{for all non-entry non-exit} \quad v \in V \setminus T^*, v \neq s^*,$$

$$f(s^*) = \sum_{(s^*,u)\in E} f(s^*,u) \quad \text{for the entry } s^*, \text{and } f(t^*) = \sum_{(u,t^*)\in E} f(u,t^*) \quad \text{for all exits } t^* \in T^*.$$

Furthermore, every vertex v with a positive count has to be reachable from s^* along the edges with positive counts, and there has to be a (directed) path from v to an exit along the edges with positive counts. We call this the *connectivity* rule. That is, $\forall v \in V$ with $f(v) > 0$

(i) there exist $s^* = v_1, \dots, v_k = v$ such that $f(v_i, v_{i+1}) > 0, 1 \leq i \leq k-1$ for some $k \geq 1$, and
(ii) there exist $v = v_1, \dots, v_k = t^*$ such that $f(v_i, v_{i+1}) > 0, 1 \leq i \leq k-1$ for some $k \geq 1, t^* \in T^*$[1].

We call the computed vertex and edge counts *valid* if they satisfy the flow conservation and connectivity rules.

[1]We note in passing that if flow conservation holds then (i) implies (ii) and vice-versa.

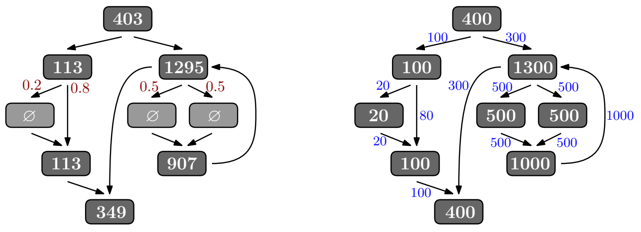


Fig. 4. An input (left) and an output (right) for the Profile Inference problem: A control-flow graph contains 9 basic block with input vertex weights (white) collected via sampling-based hardware performance counters; branch probabilities (red) are computed via a static predictive model. The output inferred counts satisfy flow conservation and connectivity rules.

Objectives. Intuitively, our goal is to infer a solution such that vertex counts are close to the input weights while edge counts respect input branch probabilities. However it is easy to see that achieving both goals simultaneously is not always possible. Therefore, we treat the first goal (respecting input vertex weights) as the primary objective and the second goal (respecting input branch probabilities) as the secondary one. The assumption is that vertex weights are a result of profiling an actual binary, while branch probabilities are likely coming from a predictive model, which is arguably less trustworthy.

In order to measure how close the computed vertex counts, denoted $f(v)$ for $v \in V$, to the input weights, $w(v)$, we introduce the following cost function:

$$cost(V) = \sum_{v \in V} cost\big(f(v), w(v)\big),$$

where the sum is taken over all non-dangling vertices of the graph, that is, when $w(v) \neq \emptyset$. Here the term $cost\big(f(v), w(v)\big)$ penalizes the change of a vertex weight from w to f; this is a monotone function with respect to their absolute difference, $|f - w|$. In our implementation we consider linear and quadratic functions, and distinguish between increasing and decreasing the weights:

$$cost_{L_1}(f, w) = \begin{cases} k_{inc}(f - w) & \text{if } f(v) \geq w(v) \\ k_{dec}(w - f) & \text{if } f(v) < w(v) \end{cases} \quad \textit{and} \quad cost_{L_2}(f, w) = \begin{cases} k_{inc}(f - w)^2 & \text{if } f(v) \geq w(v) \\ k_{dec}(w - f)^2 & \text{if } f(v) < w(v) \end{cases}$$

Here k_{inc} and k_{dec} are non-negative penalty coefficients, whose exact values are described in Section 4.

For the second objective, we measure how far are desired branch probabilities from the ones generated by an algorithm. To this end, we define $p(v, u) = \frac{f(v,u)}{f(v)}$ as the inferred probability of taking edge (v, u) from a vertex v with $f(v) > 0$ to its successor $u \in V$. For vertices with $f(v) = 0$, we define $p(v, u) = \frac{1}{\text{succs}(v)}$, where $\text{succs}(v)$ is the number of successors of v. By the flow conservation rules, we have that $0 \leq p(u, v) \leq 1$ for all edges of the graph. Now we define the objective function for preserving branch probabilities:

$$cost(E) = \sum_{(u,v) \in E} |p(u, v) - w(u, v)|.$$

Overall, our optimization problem is as follows: Given a directed graph $G = (V, E, w)$ with vertex and edge weights, find valid flow counts minimizing $cost(V)$, and among those solutions one that minimizes $cost(E)$. We call it the Profile Inference Problem (PIP). Figure 4 illustrates the input (left) and a possible output (right) for the problem.

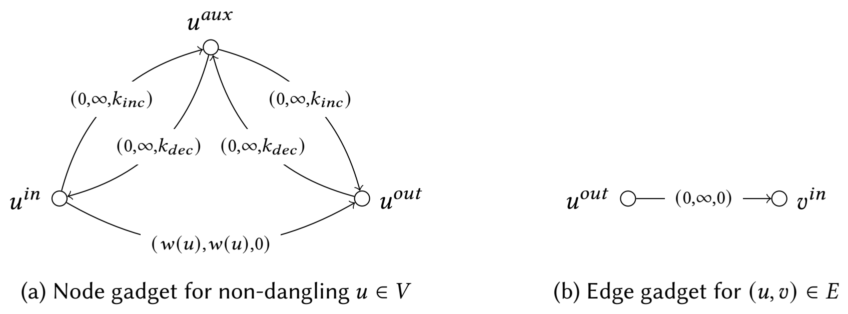


Fig. 5. Finding a flow (without connectivity constraints) via MCF. Edge labels show the lowerbound-capacity-cost triplet for the edge.

3 THEORY AND ALGORITHMS

The flow conservation rules in PIP hint that the problem could be reduced to MCF. Recall that in the MCF problem we are given a graph with a source s and a sink t, edge capacities, and edge costs. The objective is to find an s-t flow obeying flow conservation rules that sends the maximum amount of flow out of s and subject to that, one that has minimum cost $\sum_e cost(e)f(e)$. There are several polynomial time algorithms for MCF [Ahuja et al. 1993]. Some generalizations of MCF can be reduced to the basic problem, such as having piecewise convex edge costs, or lowerbound constraints on the flow that must be sent along an edge.

Indeed, such an approach has already been proposed in the literature [Levin et al. 2008; Ramasamy et al. 2008]. However, the treatment given in those papers does not capture the whole of our formulation. Next we present a theoretical framework for profile inference and show that the algorithms coming out of our framework can be engineered to work effectively at scale.

3.1 Flow Conservation

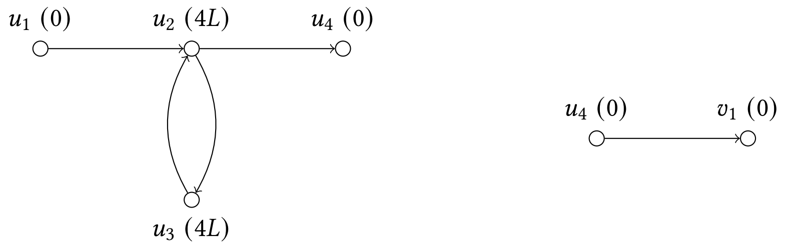
Given an instance (G, w) of PIP, we reduce the problem of finding a flow f obeying the conservation rules (but not necessarily connected) and minimizing $cost_{L_1}(f, w)$ to an MCF instance.

For each vertex $u \in V$ we create three nodes in the MCF instance: u^{in}, u^{aux} and u^{out}. For each $u \in V$ that is non-dangling, we create an edge (u^{in}, u^{out}) with capacity $w(u)$, lowerbound $w(u)$ and cost 0, edges (u^{in}, u^{aux}) (u^{aux}, u^{out}) with capacity ∞, lowerbound 0 and cost k_{inc}, and edges (u^{out}, u^{aux}) and (u^{aux}, u^{in}) with capacity $w(u)$, lowerbound 0 and cost k_{dec}. For each $u \in V$ that is dangling, we create an edge (u^{in}, u^{out}) with capacity ∞, lowerbound 0, and cost 0. For each edge (u, v) in G we add the edge (u^{out}, v^{in}). Finally, we introduce a new source s' and connect it to s^{in} with an edge of capacity $w(s)$, lowerbound 0, and cost 0; and a new sink t' and for every exit node u we connect u^{out} to t' with an edge with capacity ∞, lowerbound 0, and cost 0. Figure 5 illustrates the gadgets used in the reduction.

Let f' be an s'-t' flow in the MCF instance. We can construct a flow f for G as follows. For each edge (u, v) in G, we set $f(u, v) = f'(u^{out}, v^{in})$ and $f(u) = f(u^{in}, u^{out}) + f(u^{in}, u^{aux}) - f(u^{out}, u^{aux}) = w(u) + f(u^{in}, u^{aux}) - f(u^{out}, u^{aux})$.

Because the cycle $u^{in} \to u^{aux} \to u^{out} \to u^{aux} \to u^{in}$ has positive weight, in any minimum cost s'-t' flow we must have $f(u^{in}, u^{aux}) = 0$ and $f(u^{aux}, u^{out}) = 0$, or $f(u^{out}, u^{aux}) = 0$ and $f(u^{aux}, u^{in}) = 0$. Therefore, the cost of f' equals $2cost_{L_1}(f, w)$.

To get rid of the flow lowerbound constraints, we use a well-known reduction to the version of the problem without such constraints. First, turn the s'-t' flow problem into a circulation problem by adding an edge of infinity capacity and zero cost from t' to s'. We create new source and sink



(a) Vertex gadget for each $u \in V$ (b) Edge gadget for $(u, v) \in E$

Fig. 6. Gadgets used in the APX-hardness of PIP. Node weights are given in parenthesis.

nodes s'' and t''. Finally each edge (u^{in}, u^{out}) with lowerbound $w(u)$ is removed from the instances and edges (s'', u^{out}) and (u^{in}, t'') with capacity $w(u)$ and cost 0 are introduced. A maximum s''-t'' flow induces a circulation in the original graph (plus the (t', s') edge) that obeys the lowerbound constraints and has the same cost. Therefore, a minimum cost maximum s''-t''-flow yields the desired minimum cost flow for the PIP instance (G, w).

3.2 Connectivity

While the output of the MCF algorithm returns a flow f that obeys the flow conservation constraints, the solution needs not be connected. Figure 2a (right) illustrates such a solution. Note that such edge flow values can never be realized by any sequence of executions having the source as the entry point, since the flow cycle is disconnected from the source. Disconnected components, *islands*, can mislead the optimizer to treat live path as dead, leading to sub-optimal code layout [Newell and Pupyrev 2020]. Furthermore, some PGO implementations assume connectivity, and flow islands can lead to magnified profile inaccuracies.

It would be desirable to generalize the MCF algorithm to produce a minimum cost flow that is also connected. Unfortunately, as the next theorem shows, this problem is not only NP-hard to solve but also APX-hard to approximate[2].

Theorem 3.1. *Finding a connected flow f minimizing $cost_{L_1}(f, w)$ is* APX*-hard.*

Proof. We reduce from the s-t Path Traveling Salesman Problem (Path-TSP) in which we are given a graph $G = (V, E)$ and the objective is to find a minimum length path from s to t that visits all vertices in G, where the path sought need not be simple.

We construct an instance of PIP as follows. For each $u \in V$, introduce four nodes u_1, u_2, u_3, u_4 with weights 0, $4L$, $4L$, and 0, respectively, and connect them as follows: (u_1, u_2), (u_2, u_3), (u_3, u_2), and (u_2, u_4). For each $(u, v) \in E$, add edge (u_4, v_1). Finally, create a new source s' and add a path of length $2L - 1$ from s' to s_1 with nodes of weight 0, and a new target t' with a path of length $2L - 1$ from t_4 to t' with nodes of weight 0. Figure 6 illustrates the gadgets.

Any connected s'-t' flow can be decomposed into a collection of (not necessarily simple) s'-t' paths, which in turn induce s-t paths $P_1, \ldots, P_\ell$ in G by contracting every vertex gadget and trimming the part of the paths from s' to s_1 and from t_4 to t'. If the connected flow f is optimal, we can assume without loss of generality that $f(u_3) = 4L$ if there is a path P_i such that $u \in P_i$ and $f(u_3) = 0$. Thus, $cost_{L_1}(f, w) = \sum_i (4L + 3|P_i|) + 8L|U|$, where $U = \{u \in V : u \notin P_i \ \forall i\}$ is the set of

[2]Recall that a problem is NP-hard if a polynomial time algorithm implies $P = NP$, while an optimization problem is said to APX-hard if there exists a constant $\alpha > 1$ such that a polynomial algorithm that always returns a solution with cost at most α times the optimal cost implies $P = NP$.

vertices not spanned by any path. Finally, notice that $|P_i| \le n^2$. To see why this claim holds, split P_i into a collection of simple cycles and observe that every cycle must have a unique node; otherwise, one of those cycles can be skipped leading to a better solution.

Let P be the optimal path for the Path-TSP[3]. If we set $L = |P|$, then the optimal solution to the PIP instance induces a collection of paths covering all vertices (that is, $U = \emptyset$); since it is always cheaper to have one more path and one fewer node in U because $4L + 3|P| \le 8L$. Similarly, it is better to have a single (perhaps long) path P than two (perhaps short) paths P' and P'' covering the same set of nodes since $4L + 3|P| \le 8L \le 4L + 3|P'| + L + |P''|$.

It follows that the optimal solution for the flow problem induces a single s-t path that visits all the nodes in G and has minimum length. In other words, a polynomial time algorithm for finding an optimal connected flow can be used to solve the Path-TSP instance. Furthermore, if we have an α-approximation with $1 < \alpha < 8/7$, the solution induces a single s-t path P' such that $4L + 3|P'| \le \alpha(4L + 3|P|)$. Hence, $|P'| \le \frac{7\alpha-4}{3}|P|$. If we could make α arbitrarily close to 1 then we could approximate the Path-TSP arbitrarily close to 1 as well, which is not possible since Path-TSP is APX-hard [Papadimitriou and Vempala 2006]. □

We mention that other versions of the connect flow problem have been studied where the aim is to find a connected flow attaining a certain flow at designated vertices [Mannens et al. 2021]. Our setting is different because the islands are not prescribed in advance, and even if they were, we have the freedom of connecting any vertex of an island to the source. In that sense, finding an optimal connected flow is a harder problem than the one studied by [Mannens et al. 2021].

It follows that in practice we must resort to heuristics to connect flow islands to the source. We observe that the real-world instances of PIP do not look like the hard instances in the proof of Theorem 3.1. The optimal flow f (without connectivity constraints) can be converted into a connected flow f' by iteratively finding s-t paths that span each of the islands and pushing one unit of flow along said paths to connect all the islands. Typically, the additional cost incurred by this modification, $(cost(f', w) - cost(f, w))$, is negligible compared to $cost(f, w)$.

3.3 Quadratic Objective

While finding a connected min-cost flow is a hard problem, not all generalizations of the MCF problem are hard. In this section we consider the problem of finding a flow f minimizing quadratic objective $cost_{L_2}(f, w)$. There exists a simple pseudo-polynomial reduction of this problem to the basic linear objective not just for quadratic objectives, but for any integral convex function [Ahuja et al. 1993], and specialized algorithms that run in strongly polynomial time [Végh 2016].

The quadratic objective is the right objective to optimize if we assume the input weights, w, are obtained from some ground truth, g, by independently adding Gaussian noise to each vertex.

Theorem 3.2. *Let g be the ground truth of execution counts. If $w(u) \sim N(g_u, \sigma)$ for all $u \in V$, independent from one another, then a flow f minimizing $cost_{L_2}(f, w)$ is a maximum likelihood estimator of g.*

Proof. The following likelihood function captures how likely it would be to observe w if the ground truth was a given flow f.

$$L(f, w) = \prod_{u \in V} \Pr[w(u) | w \sim N(f, \sigma).]$$

Following the standard argument for the least-squares method in linear regressions [Feller 1968], we get that f is maximizer for $L(f, w)$ if and only if f is a minimizer for $cost_{L_2}(f, w)$. □

[3]In the reduction, we do not need to know the identity of P just its cardinality, which we can guess exhaustively in polynomial time since $|P| \le n^2$.

The noise model used in Theorem 3.2 may not be realistic in some settings but it may be a good approximation in others. For example, if w is constructed from the ground truth by uniformly sampling execution counts so that each block execution has a probability $p < 1$ of being counted in w, then $w \sim B(g_u, p)$ is a binomial distribution, which for large g_u and fixed p can be approximated with $N(g_u p, g_u p(1-p))$ when g_u is large and p is bounded away from 0 and 1. This is still not exactly the model needed for Theorem 3.2 to hold, but is a close enough fit in some cases, say when the hot parts of g are within some reasonably narrow range. In Section 5 we verify the assumptions in practice by comparing the results obtained by using linear and quadratic objectives for MCF.

3.4 From Edge Weights to Vertex Weights

We now turn our attention to the problem of inferring a flow when we lack data about execution counts but have a prior on the edge weights. This can happen when we have a region of the graph where all the blocks are dangling. Given that these blocks do not contribute to the objective, the MCF algorithm arbitrarily picks a path going through the region every time it needs to push flow through it. However, it is desirable to assign counts so as to minimize $cost(E)$.

Let us assume first that the whole graph $G = (V, E)$ is made up of dangling nodes (that is, $w(u) = \emptyset$ for all $u \in V$) but we do know branch probabilities, $w(u, v) > 0$ for all $(u, v) \in E$, that we would like to maintain. Without loss of generality we assume that there exists a path from the source to every node in the graph and that there exists a path from every node to a terminal node (one having out-degree 0). Our goal is to find a flow f out of the source node such that $p(u, v) = f(u, v)/f(u) = w(u, v)$ for all $(u, v) \in E$; that is, we would like to find a solution with $cost(E) = 0$. This can be done in $O(|E|)$ time if the graph is acyclic: Find a topological order of G and set the flow values according to

$$f(v) = \sum_{(u,v)\in E} f(u)w(u, v) \tag{1}$$

in the topological order. Thus, we focus on the more challenging cyclic case.

We now relate the problem of finding such a solution to the problem of finding a stationary solution for a certain Markov Chain M. We let the rows and columns of M be indexed by vertices in V such that $M_{u,v} = w(u, v)$ for $(u, v) \in E$ and $M_{t^*,s^*} = 1$ for all terminal nodes t^* in V and source s^*. Recall that the stationary distribution of a Markov Chain M is the vector σ such that $\sigma^T M = \sigma^T$.

Observation 1. *A flow f obeys the edge probabilities constraints if and only if $\frac{f}{||f||_1}$ is a stationary distribution of M.*

If a transition matrix is *irreducible* (that is, if we can reach any state from any state by a sequence of transitions with a positive probability), then a stationary distribution always exists and it is unique. This is certainly true for M since from any state we can reach a terminal node with positive probability, from any terminal node we can jump to the source, and from there we can reach any other state with positive probability.

There are several methods for obtaining the stationary distribution of a Markov Chain M. One method is to solve a set of $|V|$ linear equations, which can be done in $\Theta(|V|^3)$ time using Gaussian elimination, which is too slow for our purposes. Given the sparse nature of the transition matrices arising from control-flow graphs, a natural alternative is to use the power method: starting from the uniform probability distribution $x^{(0)}$ we iterate over the chain $x^{(i+1)} = x^{(i)}M$ until the difference between two consecutive solution is small enough; namely, until $||x^{(i+1)} - x^{(i)}||_1 \leq \delta$ for a given error threshold δ. Each iteration takes $O(|E|)$ time to compute. The approach converges to a solution but it might take too many iterations in practice. Through experimentation, we observed that we do not need to update the whole vector $x^{(i)}$ in locked-step. Instead, we keep a queue of nodes that need updating (initially has all nodes). In each iteration we remove a node from the queue and we update its entry using Equation 1. If the update causes the value of the node change more than a threshold δ', then we put the node and its out-neighbors back in the queue (if not there already).

The method not only applies to instances where all nodes are dangling, but also to subgraphs made up of dangling nodes that have a single entry node from the rest of the graph and a single exit node to the rest of the graph. We call these structures *dangling subgraphs*. From the point of view of $cost_{L_1}$, how the flow is routed from the entry node to the exit node of a dangling subgraph is irrelevant and their flow values do not count towards the objective. Therefore, after running the MCF algorithm, we post-process the flow using the iterative algorithm inside each dangling subgraph so that it conforms to the given branch probabilities.

4 IMPLEMENTATION AND ENGINEERING

In this section we outline a custom MCF solver that exploits the special structure of the instances that come out of the PIP-to-MCF reduction presented in Section 3.1. The solver circumvents some of the inherent shortcomings of the MCF approach by trying to distribute flow evenly between multiple paths instead of using an arbitrary path like the standard MCF algorithms. This solver is at the heart of our new algorithm, referred to as **`profi`**; see Algorithm 1.

In order to explain the details of **`profi`**, first we need to provide an overview of the Ford-Fulkerson algorithm [Ford and Fulkerson 1956] for MCF, which we build upon. Given a directed graph $G = (V, E)$ with edge capacities *cap* and edge costs *cost* and two nodes s and t, the algorithm computes a flow f obeying the capacity constraints $f(u, v) \leq cap(u, v)$ having maximum value $\sum_{(s,u)\in E} f(s, u)$, and among those solutions one that minimizes $\sum_{(u,v)\in E} cost(u, v)f(u, v)$.

The algorithm builds an auxiliary network H that has the same vertex set V. For every edge $(u, v) \in E$ in G, we have two anti-parallel edges: (u, v) is called a *forward* edge and has cost $cost(u, v)$, while (v, u) is a *backward* edge and has cost $-cost(u, v)$. The former is used to increase the flow along (u, v), while the latter is used to decrease the flow along the edge if needed. A forward edge is called *saturated* if $f(u, v) = cap(u, v)$, while a backward edge is saturated if $f(u, v) = 0$.

Starting from the empty flow, Ford-Fulkerson iteratively increases the value of the flow until arriving at a maximum flow. In each iteration it finds a minimum cost s-t path using unsaturated edges and pushes flow along the path until one of the edges in the path becomes saturated. In doing so, the algorithm maintains the invariant that current flow has minimum cost among all flows that have the same value. Crucially, any such path or combination of such paths can be used in this update without sacrificing the optimality of the final flow the algorithm returns. Our method **`profi`** takes advantage of this property to speed up the running time of the algorithm and to improve the quality of the inferences found. We describe these two ideas in the next sections.

4.1 Improving the Running Time

The overall running time of our MCF solver is dictated by two quantities: The number of augmentations (number of iterations) and the average time spent per iteration. We consider improvement to each of these quantities separately.

Number of iterations. We note that although the number of iterations of the Ford-Fulkerson can be as large as the value of the maximum flow[4], the fact that the instances in our reduction only have edges with finite capacity out of s'' or into t'' means the absolute worst case number of iterations is at most $O(|V||E|)$. To see this, observe that we cannot have more than $O(|E|)$ consecutive iterations without saturating and edge incident on s'' or t'', and there are only $2|V|$ such edges. And once those edges are saturated they remain saturated.

[4]The maximum value of the flow in our instances is $\sum_{u\in V} w(u)$, which would make the algorithm prohibitively expensive.

Algorithm 1: profi

```
Input  : control-flow graph $G = (V, E)$ with the entry $s^* \in V$,
         vertex sample counts $w : V \to \mathbb{N} \cup \{\varnothing\}$,
         branch probabilities $w : E \to \mathbb{R}$
Output : valid flow along vertices and edges $f : V \cup E \to \mathbb{N}$

Function ProfileInference
    /* create auxillary flow network                                  */
    $(H, s'', t'') \leftarrow$ AuxillaryNetwork$(G, s^*)$;
    while FindAugmentingDAG$(H, s'', t'')$ do              /* flow augmentation */
        AugmentFlowAlongDAG$(H, s'', t'')$;
    /* post-processing adjustment of the flow                         */
    JoinIsolatedComponents$(H)$;
    RebalanceDanglingSubgraphs$(H)$;
    return flow values given by $H$;

Function FindAugmentingDAG$(H, s'', t'')$
    $d \leftarrow$ ModifiedMooreBellmanFord$(H, s'', t'')$;
    if $\nexists$ $s''$-$t''$ path then
        return false;
    $X \leftarrow \{(u, v) \in H : d(v) = d(u) + w(u, v)\}$;
    $X \leftarrow$ ExtractDAG$(X, s'', t'')$ ;
    return true;

Function ModifiedMooreBellmanFord$(H, s'', t'')$
    $d(s'') \leftarrow 0$  and  $d(v) \leftarrow \infty$ for $v \neq s''$;
    $Q \leftarrow$ queue holding vertex $s''$ ;
    while $|Q| > 0$ and $d(t'') > 0$ do
        $u \leftarrow$ pop$(Q)$ ;
        if $d(u) \leq d(t'')$ then
            for $(u, v) \in H$ do
                if $d(u) + w(u, v) < d(v)$ then
                    $d(v) \leftarrow d(u) + w(u, v)$ ;
                    add $v$ to $Q$ if not there already;
    return $d$;

Function AugmentFlowAlongDAG$(H, s'', t'')$
    $\Delta(s'') \leftarrow$ target flow update value  and  $\Delta(u) \leftarrow 0$ for $u \in V - s$;
    for $u \in V$ in topological order of $X$ do   /* $X$ is DAG found in FindAugmentingDAG */
        set $\Delta(u, v)$ for $(u, v) \in X$ by distributing $\Delta(u)$ as evenly as possible;
        $\Delta(v) \leftarrow \Delta(v) + \Delta(u, v)$ for $(u, v) \in X$;
    push $\Delta(u, v)$ units of flow along every $(u, v) \in X$;
    update auxiliary graph $H$ accordingly;
```

To further reduce the number of iterations, we note that we can greedily augment our flow along short augmenting paths of zero cost. There are many such paths. In fact, for each edge $(u, v) \in G$ in the PIP instance, we have a path $s'' \rightarrow u^{out} \rightarrow v^{in} \rightarrow t''$. Augmenting along these paths does not increase the cost of the flow (as these edges have zero cost) but increases the value of the flow thus ultimately reducing the number of iterations of the algorithm.

Iteration complexity. In each iteration, we need to find a shortest s''-t'' path in the auxiliary network H. We use a modified version of the classical Moore-Bellman-Ford [Bellman 1958; Ford 1956; Moore 1959] algorithm for finding a shortest path in a graph with positive and negative weights. The worst-case complexity of the algorithm is $O(|V||E|)$ but its typical complexity is close to linear. We further improve this by pruning the search space of parts of the graph that reduce the running time. To explain our contribution, we first discuss the details of the classical algorithm.

The original Moore-Bellman-Ford algorithm maintains a distance label $d(u)$ for each $u \in V$. Initially $d(s'') = 0$ and $d(u) = \infty$ for $u \in V - s''$. The algorithm maintains a queue Q, which initially holds s. In each iteration, we remove the node u at the front of Q and we attempt to update the distance labels of the out-neighborhood of u by setting $d(v) = \min(d(v), d(u) + w(u, v))$ for each $(u, v) \in E$ that is not saturated. If the update reduces the distance label of v, then we add v to Q if not there already.

Our refinement is not to update the out-neighborhood of u if $d(u) > d(t'')$ and to cut the shortest path computation short once $d(t'') = 0$. We observe in our experiments that this refinement greatly speeds the algorithm. We make use of the following property of shortest-path distances in H. Here $\text{dist}(u, v)$ is the cost of the shortest u-v path in H using non-saturated edges.

Lemma 4.1. *For any $v \in H$, $\text{dist}(s'', v) \geq 0$ and $\text{dist}(v, t'') \geq 0$.*

Proof. We now argue that $\text{dist}(v, t'') \geq 0$ from any vertex v; the argument that $\text{dist}(s'', v) \geq 0$ follows along the same lines. Let v be a vertex that minimizes $\text{dist}(v, t'')$ and subject to that minimizes the number of edges in the path from v to t'' realizing that distance. Assume, for the sake of contradiction, that $\text{dist}(v, t'') < 0$ and let P be the path realizing this distance.

Now suppose that there exists a path Q from v to t'' in the original graph G such that $f(e) > 0$ for all $e \in Q$. Notice that for each $e \in Q$ we have $cost(e) \geq 0$ and its anti-parallel version is non-saturated. Therefore, if we reverse the direction of Q we get a path $\bar{Q}$ from t'' to v that is non-saturated and has $cost(\bar{Q}) \leq 0$. Splicing P and $\bar{Q}$ together, we get a negative cost cycle[5] that we could use to reduce the cost of the flow without changing the amount of flow being shipped from s'' to t'', which would contradict the main invariant of the Ford-Fulkerson algorithm.

Now suppose that there is no such path Q. Let u be the vertex after v in P. If $cost(v, u) \geq 0$ then $\text{dist}(u, t'') \geq \text{dist}(v, t'')$ and u attains that distance with fewer edges, which would contradict the way we picked v. Otherwise, $cost(v, u) < 0$. This means that (v, u) is a backward edge, so $f(u, v) > 0$. The only way we can have flow coming into v but no flow path from v to t'' is that v and u belong to a flow island. This means that there must be a cycle C such that $(u, v) \in C$ and $f(e) > 0$ for all $e \in C$. Let $\bar{C}$ be the anti-parallel version of C. Notice that $\bar{C}$ is a non-saturated cycle in H and that $cost(\bar{C}) \leq cost(v, u) < 0$. Therefore, we can improve the cost of the flow by pushing flow around $\bar{C}$ without changing the amount of flow being shipped from s'' to t'', which would again contradict the main invariant of the Ford-Fulkerson algorithm.

We have reached a contradiction in every case, so we conclude that $\text{dist}(v, t'') \geq 0$ for all v. □

The following lemma argues that this faster algorithm is good enough for the purposes of the correctness of **profi**.

[5]This cycle need to be simple; if an edge is present in P and Q then its multiplicity when computing the cycle's cost needs to be taken into account accordingly.

Lemma 4.2. *Let d be the output of Modified-Moore-Bellman-Ford. If $d(t'') > 0$ then for every v that lies on a shortest s''-t'' non-saturated path in H we have that* $\text{dist}(s'', v) = d(v)$.

Proof. To prove the claim, let us assume, for the sake of contradiction, that there exists a node v on a shortest s''-t'' non-saturated path in H such that $\text{dist}(s'', v) < d(v)$ and let v be such a node with the shortest path having the fewest edges. Let u be the vertex before v along this path. It follows that $\text{dist}(s'', u) = d(u)$ and $\text{dist}(s'', v) = d(u) + w(u, v)$. Consider the last time we took u out of the queue Q. At this point in time, we must have had $d(u) > d(t'')$, for otherwise, we would have updated the label of v to $d(u) + w(u, v) = dist(s'', v) < d(v)$. On the other hand, the following derivation implies the opposite

$$d(u) = \text{dist}(s'', u) = \text{dist}(s'', t'') - \text{dist}(u, t'') \leq \text{dist}(s'', t'') \leq d(t''),$$

where first inequality follows from the $\text{dist}(u, t'') \geq 0$ for all u, and the second inequality from an invariant of the classical Moore-Bellman-Ford algorithm.

We have reached a contradiction in every case, so we conclude that $\text{dist}(s'', v) = d(v)$ and the claim follows. □

Lemma 4.3. *Let d be the output of Modified-Moore-Bellman-Ford. If $d(t'') = 0$ then there exists a shortest s''-t'' non-saturated path in H such that for every v in this path we have that* $\text{dist}(s'', v) = d(v)$.

Proof. Since $\text{dist}(s'', v) \geq 0$, once the algorithm updates the distance label of t'' to $d(t'') = 0$, we know that $dist(s'', t'') = d(t)$, so we can cut the search off. □

We stress that this lemma holds for any MCF instance with non-negative edge weights.

Typical time complexity. These optimizations mean that on typical instances we need fewer than $|V|$ iterations and that each iteration takes less than linear time to execute, usually yielding an observed running time that is sub-quadratic, that is, $o(|V|^2)$. In Section 5.4 we give a more detailed experimental analysis of the running time of **`profi`**.

4.2 Finding the Augmenting DAG

Another refinement we introduce to our MCF solver is the use of multiple shortest paths instead of a single shortest path. While this does not affect the complexity, it avoids lopsided inferences where one path in the input is arbitrarily preferred over another one of equal cost.

We implement this by identifying all edges that belong to some shortest path out of s''. This is easy to do using the shortest-path labels computed in the auxiliary graph:

$$X = \{(u, v) \in H : d(v) = d(u) + cost(u, v)\}.$$

The set X is pruned by keeping those edges that lie on an s''-t'' path and further pruned by removing edges to break cycles (if any) while maintaining s''-t'' connectivity among the remaining edges. This pruning task can be done by computing the strongly connected components of X, removing edges within each component until they are acyclic, and maintaining all edges across components. All this is done using Tarjan's linear time algorithm [Tarjan 1972] for computing strongly connected components that performs a single depth-first search over the subgraph.

Let X be the result of the pruning operation; notice that it is a directed acyclic graph (DAG). We identify a target amount $\Delta(s'')$ of flow that we would like to send out of s''. This amount is propagated along the DAG by distributing the flow as evenly as possible at each node in the DAG. The value $\Delta(s'')$ is chosen so that no edge capacity is violated and at least one edge is saturated after the update, which can be done first sending one fractional unit of flow where the flow is split perfectly, call this $\delta(\cdot)$ and the setting $\Delta(s'') = \min_{e \in X} \widehat{cap}(e)/\delta(e)$, where $\widehat{cap}(e)$ is the current residual capacity along the edge, that is, how much more flow we can push along e.

4.3 Implementation in LLVM

We implemented **`profi`** in the open-source compiler LLVM as a part of the improved version of the LLVM's AutoFDO [He and Yu 2020]. AutoFDO is an implementation of sampling-based PGO; it leverages the Linux perf tool to collect a raw binary-level profile. The raw profile is then converted to a source-level profile that is consumed by the compiler to guide its optimizations.

In our implementation, a binary-level profile is generated based on the LBR traces collected using the Intel PMU hardware. An LBR trace usually consists of 16 or 32 consecutive branch records; it reflects a list of the most recently executed branch instructions. Two consecutive branch records identify a sequence of executed instructions. A binary-level profile is formed by extracting all such instruction sequences from the LBR traces collected under a certain sampling rate and duration. An IR-level (source) profile can then be generated by mapping the binary instruction sequences to the compiler IR. Unlike the default AutoFDO [Chen et al. 2016] that relies on the debug line number information to map instruction sequences to compiler IR, we use an improved approach that leverages so-called pseudo-instrumentation for a more precise binary-IR mapping.

Various compiler optimization passes in LLVM can benefit from profile data, including critical machine-independent optimizations such as function inlining, devirtualization, loop optimizations and machine-dependent transforms like register allocation and machine code layout. The PGO framework in LLVM, underpinning both instrumentation-based PGO and sampling-based PGO, is set up uniformly in the form of IR annotations so that any pass can access or update the profile counts on demand. Profile inference is positioned at the beginning of the optimization pipeline to benefit as many optimization passes as possible.

5 EXPERIMENTAL EVALUATION

To validate the effectiveness of profile inference, we (i) evaluate the precision of the inferred block and edge frequencies with respect to a correct profile, and (ii) compare the performance of binaries generated with the use of the new profile data. We also investigate the scalability of our algorithm.

The experiments presented in this section were conducted on a Linux-based server with a dual-node 28-core 2.4 GHz Intel Xeon E5-2680 (Broadwell) having 256GB RAM. The algorithms are implemented on top of `release_12` of LLVM.

5.1 Benchmarks

We evaluated our approach on both open-source workloads and very large binaries deployed at Facebook's data centers; see Table 1. As publicly available benchmarks, we used 15 C/C++ programs from SPEC CPU 2017, and two open-source compilers, Clang and GCC, utilizing their `10` and `8.3` releases, respectively. For data-center workloads, we selected three binaries. The first system is the HipHop Virtual Machine (HHVM), that serves as an execution engine for PHP; for our experiments, we utilized dynamically compiled code of the binary [Ottoni 2018]. The second is Cinder, which is Instagram's Python interpreter. The third one is Mysql, which is a database serving for a large number of Facebook products.

5.2 Techniques

We implemented **`profi`** as described in Section 4. In order to derive target branch probabilities, we use static prediction of Wu and Larus [1994] that estimates probabilities for every branch in a binary function. The algorithm, **`profi`**, is compared with the following competitors.

- **`mcf`** is the classical algorithm suggested by Levin, Newman, and Haber [2008], which is based on solving a variant of the MCF problem. Our implementation closely follows a description given in Section 3.1; refer to [Ramasamy et al. 2008] for the original implementation details.

Table 1. Basic properties of evaluated binaries

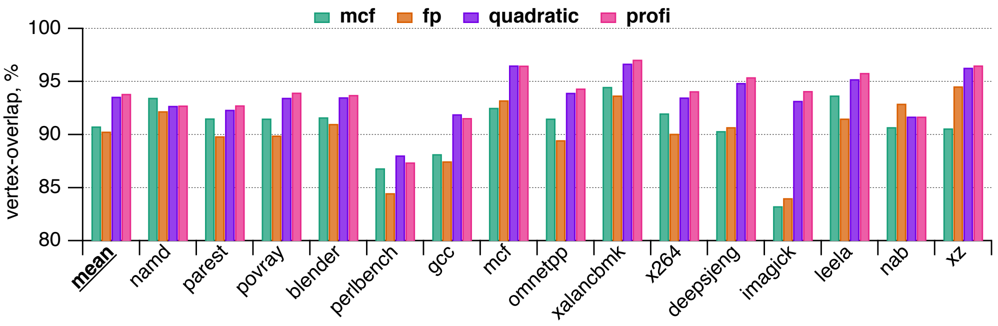
	hot code	hot	blocks per function		
	(MB)	functions	p50	p95	max
SPEC17		2,144	11	123	5,531
Clang	17	12,079	28	253	12,555
GCC	8	5,204	29	251	3,354
HHVM	21	6,832	70	362	2,406
Cinder	2	457	8	58	2209
Mysql	14	1,418	10	84	3,952

- **fp** is the frequency propagation heuristic utilized by [Chen et al. 2016] and [Novillo 2014] in the context of AutoFDO. The heuristic consists of two phases. First it identifies equivalence classes of basic blocks using dominance and loop information; the blocks in the same equivalence class always have equal weights. The second phase is an iterative process that propagates block weights into edges using three rules. If a block has a single predecessor/successor, then the weight of that edge is the weight of the block. If all the predecessor/successor edges of a block are known except one, the weight of the unknown edge is assigned to the weight of the block minus the sum of all the known edges. Finally, if there is a self edge, the weight for that edge is set to the weight of the block minus the weight of the other incoming edges to that block. We emphasize that due to its simplicity, the heuristic has been re-implemented in other compilers such as [Ottoni 2018; Ottoni and Liu 2021].
- **quadratic** is a variant of **profi** in which a quadratic cost function, $cost_{L_2}(f, w)$, is utilized; refer to Section 3.3 for details.

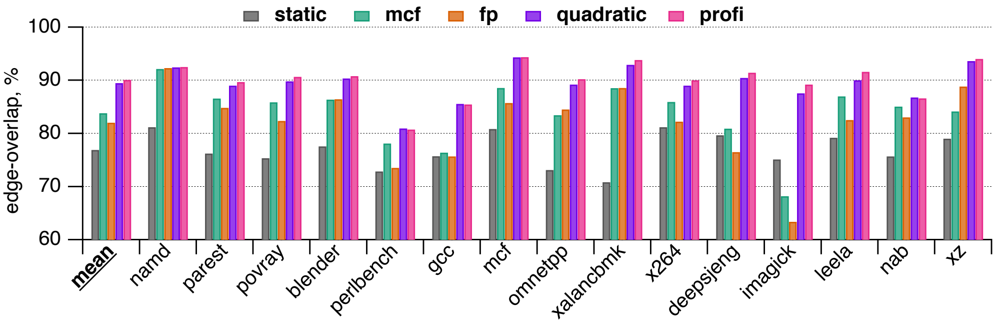
Observe that flow-based algorithms (**mcf**, **profi**, and **quadratic**) contain various parameters that can potentially impact the quality of their results. As a preliminary step, we tuned the parameters of each of the algorithms using so-called *black-box optimizer* [Letham et al. 2019]. This is a powerful tool for optimizing functions with computationally expensive evaluations; it is based on Bayesian optimization and is able to compute values for a collection of parameters that maximize a desired objective. In our setting, we seek to optimize the vertex overlap function (as defined in the next paragraph) between the ground truth and the output of the inference algorithm varying its parameters on SPEC2017 benchmark.

The primary parameters of **profi** and **quadratic** are penalty coefficients, k_{inc} and k_{dec}, introduced in Section 2.2. In our evaluation we in addition distinguish between coefficients for increasing counts for blocks whose weights are zero (denoted k^0_{inc}) and non-zero in the input. Throughout the experiments, we use the following best identified combination: $k_{inc} = 10$, $k_{dec} = 20$ and $k^0_{inc} = 11$. Intuitively, it is twice more expensive to decrease a block count than to increase it. In addition, increasing counts of cold blocks (having zero weight) is slightly more expensive than for hot ones; recall that dangling blocks have no associated input weights and are not penalized in our model.

For **mcf**, we tried to re-tune the parameters suggested in the original paper Levin et al. [2008], but found with the black-box optimizer that tuning of the parameters have a mild impact on the quality of the inferred counts computed by the algorithm. The only notable difference is that we reduced by a factor of 10 the importance of statically predicted edge frequencies in comparison with the input block weights. This aligns with our assumption that statically predicted edge frequencies and branch probabilities are less trustworthy than sampling-based basic block counts.



(a)



(b)

Fig. 7. The quality of profile counts inferred by various algorithms on SPEC2017 binaries compiled with O3+LTO mode. The overlap values are computed with respect to the ground-truth instrumentation counts.

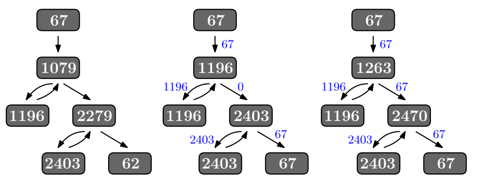
5.3 Counts Quality

To evaluate the accuracy of a profile inference algorithm, we need to compare its output to a *ground-truth* instrumentation-based profile. To this end, we evaluate the deterministic SPEC17 benchmark, where correct basic block and jump counts can be derived despite the large overhead caused by instrumentation. Following earlier works we use the *degree of overlap* measure to evaluate the quality of inferred block and edge frequencies. The measure is used to compare the similarity of two profiles with respect to a common control-flow graph. Let $f(v)$ and $f(e)$ be the inferred vertex and edge counts for $v \in V, e \in E$, respectively. Similarly, let $gt(v)$ and $gt(e)$ be the ground-truth counts for $v \in V, e \in E$. The *vertex-overlap* is defined as

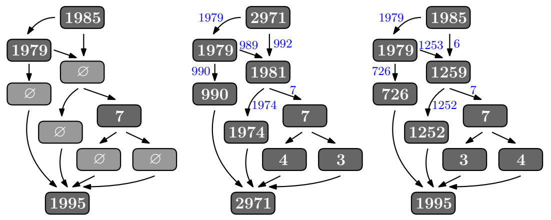
$$\sum_{v \in V} \min\left(\frac{f(v)}{\sum_{v \in V} f(v)}, \frac{gt(v)}{\sum_{v \in V} gt(v)}\right),$$

where the sum is taken over all vertices of the graph. The *edge-overlap* measure is defined similarly. If the counts in the two profiles match exactly, the overlap is equal to 1 (or 100 percent); otherwise, the measure takes values between 0 and 1.

First we evaluate the accuracy of profiles collected and inferred on binaries built in `-O0` mode, that is, with most compiler optimizations turned off. This is an easy setting in which the profiled and the processed binaries are very similar to each other, and all inference algorithms are expected to perform well. This is confirmed by a low number of dangling blocks in control-flow graphs, which constitute less than 1% of all blocks on average. The mean vertex-overlap of all tested



(a) An instance of PIP from 644.nab_s: (left) input sample vertex counts, (middle) output of **fp** with a flow island, (right) output of **profi**.



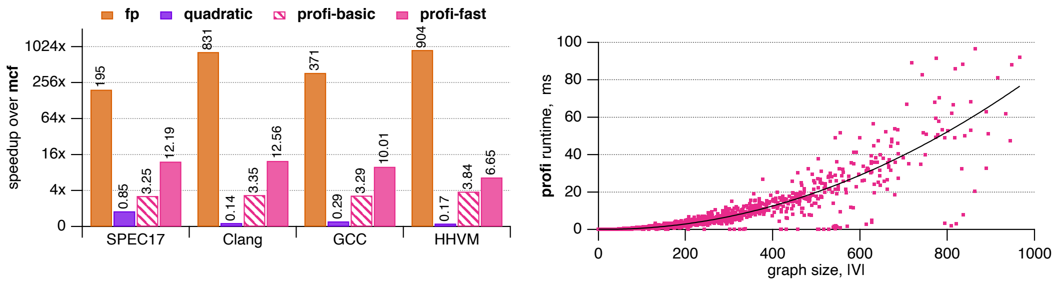
(b) An instance of PIP from 631.deepsjeng_s with equal probabilities of all branches: (left) input sample vertex counts, (middle) output of **mcf** respecting the branch probabilities at the cost of increasing the count of the entry block, (right) output of **profi**.

Fig. 8. Hard instances for existing approaches for profile inference from the SPEC17 benchmark

algorithms is over 94% across all instances, indicating that the inferred counts are close to the ground truth. However, a real-world scenario is when a binary is built in an optimized -O3 mode with an additional LTO support. Figure 7a shows the vertex-overlap between the inferred and the ground-truth instrumentation counts. We record an average of 32% of dangling blocks in the O3+LTO setting. Comparing the four inference algorithms, **profi** and **quadratic** perform the best, achieving 93.85% and 93.60% mean vertex-overlaps, respectively. The classical **mcf** yields a vertex-overlap of 90.78%, followed by **fp** producing on average a 90.29% overlap. The results are consistent with edge-overlap values presented in Figure 7b, where in addition we estimate the quality of the profile generated using static branch prediction [Wu and Larus 1994]. Again, **profi** yields the best results, improving upon the state-of-the-art **mcf** by 6.2% (from 83.8% to 90.1% across all instances). We stress that there is still a gap of around 10% to the ground truth, which leaves an opportunity for further improvements of profile data used in sampling-based PGO.

In addition to overlap measures, we analyze zero-count errors, which have been identified as an important factor influencing the effectiveness of PGO [Wu et al. 2013]. Such an error happens when an inferred block count is zero but its value in the ground-truth is non-zero. Formally, we define *coverage* as the percentage of blocks with non-zero frequency in the ground truth that also have non-zero values in the inferred profile. The measure shows how well the sampled profile represents the coverage pattern of the ground-truth profile. The coverage values computed for the algorithms are 99.18%, 98.45%, 96.98%, and 90.49% for **profi**, **quadratic**, **mcf**, and **fp**, respectively. It indicates that the new inference algorithm predicts with high accuracy hot basic blocks in instances with many dangling blocks.

In order to understand the source of the gains of **profi**, we analyze several instances with low overlap scores achieved by alternative algorithms; see Figure 8 for two such examples. Figure 8a is an instance from the 644.nab_s binary in which the result of **fp** (middle in the figure) forms an invalid flow; the algorithm, being a heuristic, is not guaranteed to produce connected components. Figure 8b is a control-flow graph from 631.deepsjeng_s containing several dangling blocks. Since the counts of such blocks are not available, an inference algorithm should respect given branch probabilities, which are all equal in the instance. The classical **mcf** achieves this at the cost of significantly increasing the count of the entry block. The new method obtains more natural counts.



(a) Ratio of runtime of **`mcf`** over runtime of various profile inference techniques on four largest datasets

(b) The runtime (in milliseconds) of **`profi`** as a function of the number of blocks measured on HHVM

Fig. 9. Processing times of profile inference

5.4 Running Time

Here we evaluate the runtime of **`profi`** in comparison with existing approaches for profile inference. To this end, we selected four largest datasets from our benchmarks, and measured the running time of each algorithm by taking a mean value over five runs on each instance. As a baseline, we utilize the classical **`mcf`**, which solves the minimum-cost maximum flow circulation problem by the so-called *cycle canceling* algorithm [Goldberg and Tarjan 1989]. The chart in Figure 9a presents relative speedups of **`profi`**, **`quadratic`**, and **`fp`** over **`mcf`**. In the figure we distinguish two variants of **`profi`**: the basic version is based on our model for MCF excluding techniques suggested in Section 4.1, while the fast version utilizes all performance improvements. The data suggests that even the basic version of our algorithm, **`profi-basic`**, significantly outperforms **`mcf`**. We attribute the speedup to the utilized Moore's algorithm for finding augmenting paths, which seems to be a good fit for PIP instances. Combining the results with optimizations described in Section 4.1, we record up to an $12x$ speedup over the baseline. Yet this is noticeably slower than the performance of the **`fp`** heuristic, which is consistently faster than all the competitors on all our benchmarks. The efficiency comes at the cost of worsen and often inconsistent inferred counts, as discussed in Section 5.3. Finally, we emphasize that the running times of **`quadratic`** noticeably exceed those of **`mcf`** in our experiments. Given that the quality of inferred profiles are superior to the ones generated by **`profi`**, we do not consider **`quadratic`** as a practical alternative.

The absolute running times of **`profi`** are illustrated in Figure 9b. While the observed complexity of the algorithm is super-linear in the number of basic blocks in a graph, the runtime does not exceed 0.1 second even for instances containing $|V| = 1000$ blocks. Since the majority of real-world instances contain much fewer vertices (see Table 1), **`profi`** is unlikely to introduce extra runtime overhead to the existing build process of data-center applications.

5.5 Binary Performance

Table 2 presents a performance comparison of binaries built with sampling-based PGO combined with four inference algorithms and with instrumentation-based PGO. All binaries are compiled using Clang (`release_12`) with `-O3` and `LTO`. For the baseline, we do not utilize PGO. To build optimized versions of the binaries, we employ a two-step approach: the first step is collecting profile data on the baseline, while the second step is re-compiling the binary with PGO enabled. In our early experiments we noticed high measurement noise that often hides the difference between the variants of the binaries. For this reason we focus on the performance evaluation of two Facebook applications, Cinder and Mysql, that can be profiled and monitored with high accuracy; in

Table 2. Performance improvements of sampling-based PGO combined with various inference algorithms and instrumentation-based PGO over binaries compiled in O3+LTO mode. The results are mean speedups (in percent) over multiple runs together with 95% confidence intervals. The best values achieved by sampling-based PGO in every column are highlighted.

	Cinder-1	Cinder-2	Mysql-1	Mysql-2	Clang-1	Clang-2	GCC-1	GCC-2	SPEC17
mcf	17.3±1.3	23.9±0.7	9.6±0.6	10.3±0.4	18.9±0.4	10.3±0.7	7.4±0.4	7.3±0.3	5.6±6.2
fp	16.5±0.3	22.9±0.3	8.9±0.5	9.1±0.4	20.3±0.3	10.9±0.6	8.3±0.3	8.2±0.2	7.2±4.9
quadratic	19.1±0.6	**26.3±0.7**	**10.2±0.6**	**10.6±0.4**	18.6±0.3	10.3±0.7	8.0±0.4	8.4±0.3	7.6±4.9
profi	**22.4±0.8**	**26.8±0.5**	**10.7±0.6**	**10.9±0.4**	**21.9±0.3**	**12.1±0.6**	**8.8±0.4**	**9.0±0.3**	7.3±4.8
instr-PGO	14.1±1.1	16.2±0.9	12.2±0.8	12.9±0.9	30.5±0.5	16.5±0.5	11.4±0.4	12.3±0.4	8.4±5.7

addition, we evaluate the performance of Clang, GCC, and SPEC17. We selected two representative benchmarks for each of the binaries. In order to further reduce the impact of measurement noise, the results are obtained by running the same binary on each workload multiple times. The table reports mean relative speedups over the runs along with their 95% confidence intervals.

First, we observe that Cinder is particularly friendly for profile-guided optimizations that achieve a double-digit performance boost over its non-optimized version. With the help of the new inference algorithm, the mean speedup is 24.5%, exceeding competitors by 3.5%. To understand the source of improvements, we analyzed the frequencies of basic blocks in the binary and how they affect various compiler optimizations. We found a single large function in the binary that is responsible for the majority of execution cycles; knowing the correct block and jump counts helps the compiler to properly layout basic blocks and make efficient inlining decisions, which greatly reduces the number of I-cache misses and branch mispredictions. The performance improvement for Mysql is more modest but still significant considering that the application is deployed in large data-centers. For the binary, flow-based approaches, **profi**, **quadratic** and **mcf**, outperform **fp** heuristic, making it faster by 1.8%. Similar speedups are observed for Clang and GCC, where **profi** outperforms the best competitor by 1.4% and 0.6%, respectively.

Next we consider SPEC17 binaries and the impact of profile inference on their performance. We do not see a consistent benefit from applying advanced inference techniques, and the overall speedup of sampling-based PGO ranges from around 25% (for `605.mcf_s` and `623.xalancbmk_s`) to a regression of 5% (for `641.leela_s`). Our investigation reveals that, despite having a better profile, various compiler optimization passes may produce sub-optimal results, as they are likely designed and over-tuned for the "imprecise" profile data.

Finally, we compare the impact of sampling-based and instrumentation-based PGO. Overall, we see a positive correlation between the quality of profile data and the performance. Mysql and GCC binaries generated with ground-truth profile are faster by 2% – 3%; for Clang, the speedup is larger. The experiment suggests that further improving profile inference might be beneficial. A notable exception is Cinder in which instrumentation-based PGO yields a regression in comparison with **profi**; we noticed the same behavior for one of the SPEC17 binaries, `605.mcf_s`. Our investigation reveals that the regression is related to the function inlining heuristic currently employed in LLVM. To drive early inlining, the compiler falls back to the total count of the callee rather than the count of the call site, since the call site count might not be reliable. However, the total callee count does not reflect runtime cost of the call, and a more accurate call site count does not lead to a better performance without modifying the inlining heuristic. We conclude that in order to realize maximum gains, the task of improving the profile quality must be combined with improving profile-guided optimizations consuming the data.

6 RELATED WORK

In this section we review previous works on accuracy enhancement techniques for sampling-based profiling. As mentioned in Section 2.1, Levin, Newman, and Haber [2008] first highlighted the importance of the problem and suggested a model based on MCF. Follow-up works utilized the flow-based algorithms for various PGO systems [Chen et al. 2016, 2013; Novillo 2014; Ottoni 2018; Panchenko et al. 2019; Ramasamy et al. 2008]. In our paper, we extend the optimization model and for the first time study its theoretical and algorithmic aspects.

Later works look at the problem from a practical point of view. Chen et al. [2013] propose to mitigate hardware sampling bias by using additional performance counters with the help of supervised learning. Subsequently, Zhou et al. [2016] present a systematic study of how sampling rates affect the accuracy of collected profiles for modern PGO. Unfortunately, as Wu et al. [2013] show, varying the sampling rate does not significantly improve the accuracy of profiling results. Liu et al. [2016] apply a machine learning approach combined with a heuristic for MCF, while Lee [2015] implements a timer-based adaptive sampling with a relatively low overhead. Alternatively, Nowak et al. [2015] utilize LBRs for instruction profiling and experimentally verify that the approach is superior over alternative techniques. However, not all processor vendors provide such logging functionality, and moreover, recent studies [Xu et al. 2019; Yi et al. 2020] indicate that LBRs suffer from sampling bias. The results hint that a post-processing step correcting anomalies in the sampling-based profiles is needed, and our approach is complementary to the above methods.

Finally we mention that due to their rich combinatorial structure and many applications, maximum flow and circulation problems have been (and continue to be) studied extensively in the Computer Science and Operations Research communities. The book by Ahuja, Magnanti, and Orlin [1993] is an excellent reference source for the theory and applications of maximum flow.

7 DISCUSSION AND CONCLUSION

In this paper we extended the state-of-the-art model for profile inference, which is an essential step in sampling-based profile-guided optimizations. We formalized the model and took a first step towards a solid theoretical foundation of the corresponding optimization problem. We also performed an extensive evaluation of existing profile inference techniques on a variety of real-world applications. The experiments indicate that the new technique significantly improves the quality of utilized profile data, reducing the gap between sampling-based PGO and its instrumentation counterpart. There are several possible extensions of our work that we discuss next.

From a theoretical point of view, PIP is a computationally hard problem and, in its general form, does not admit good approximations. However, the real-world instances arising in applications exhibit certain structural properties, which may lead to algorithms with provable guarantees. For example, the control-flow graphs of structured goto-free programs in many programming languages have constant *treewidth*, which is a standard concept in graph theory to measure the closeness of a graph to a tree [Chatterjee et al. 2019; Mestre et al. 2021; Thorup 1998]. Since many NP-hard optimization problems can be solved efficiently on graphs with a small treewidth, it is reasonable to explore PIP parameterized by the treewidth of the input graph. Another interesting direction is to develop a more realistic noise model than that outlined in Section 3.3.

From a practical point of view, there is still a performance gap between binaries processed by sampling-based and instrumentation-based PGO in our implementation. Furthermore, as we have observed in the experiments, the performance of some binaries generated with a new profile regresses. Our preliminary investigation reveals that some of existing optimization passes in LLVM (e.g., function inlining) might be over-tuned for the "imprecise" profile data. Re-tuning existing heuristics employed in the compiler is a necessary future step.

REFERENCES

Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, Inc., USA.

Thomas Ball and James R Larus. 1994. Optimally profiling and tracing programs. *Transactions on Programming Languages and Systems* 16, 4 (1994), 1319–1360.

Richard Bellman. 1958. On a routing problem. *Quart. Appl. Math.* 16 (1958), 87–90.

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 Transactions on Programming Languages and Systems* 19, 1 (1997), 188–222.

Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019. Efficient parameterized algorithms for data packing. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28.

Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *International Symposium on Code Generation and Optimization.* ACM, 12–23.

Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. 2013. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Trans. Comput.* 62, 2 (2013), 376–389.

William Feller. 1968. *An Introduction to Probability Theory and Its Applications.* Wiley.

L. R. Ford. 1956. *Network Flow Theory.* RAND Corporation, Santa Monica, CA.

L. R. Ford and D. R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

Andrew V. Goldberg and Robert E. Tarjan. 1989. Finding minimum-cost circulations by canceling negative cycles. *J. ACM* 36, 4 (1989), 873–886.

Wenlei He and Hongtao Yu. 2020. [llvm-dev] [RFC] Context-sensitive Sample PGO with Pseudo-Instrumentation. https://lists.llvm.org/pipermail/llvm-dev/2020-August/144101.html.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization.* IEEE, 75.

Byeongcheol Lee. 2015. Adaptive correction of sampling bias in dynamic call graphs. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2015), 1–24.

Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. 2019. Constrained Bayesian optimization with noisy experiments. *Bayesian Analysis* 14, 2 (2019), 495–519.

Roy Levin, Ilan Newman, and Gadi Haber. 2008. Complementing missing and inaccurate profiling using a minimum cost circulation algorithm. In *High Performance Embedded Architectures and Compilers*, Vol. 4917. Springer, 291–304.

Xian-hua Liu, Yuan Peng, and Ji-yu Zhang. 2016. A sample profile-based optimization method with better precision. In *International Conference on Artificial Intelligence and Computer Science.* DEStech Publications, 340–346.

Isja Mannens, Jesper Nederlof, Céline Swennenhuis, and Krisztina Szilágyi. 2021. On the parameterized complexity of the connected flow and many visits TSP problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, Vol. 12301. Springer, TBA.

Julián Mestre, Sergey Pupyrev, and Seeun William Umboh. 2021. On the Extended TSP Problem. In *Proc. of the 32st International Symposium on Algorithms and Computation.*

Edward F. Moore. 1959. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*. Harvard Annals, Harvard University, 285–292.

Andy Newell and Sergey Pupyrev. 2020. Improved basic block reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794.

Diego Novillo. 2014. SamplePGO: the power of profile guided optimizations without the usability burden. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC.* IEEE Press, 22–28.

Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a base of trust with performance counters for enterprise workloads. In *USENIX Annual Technical Conference.* USENIX Association, 541–548.

James Orlin. 1988. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*. ACM, 377–387.

Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 151–165.

Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting both warmup and steady-state performance at scale. In *International Symposium on Code Generation and Optimization.* IEEE, 340–350.

Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: a practical binary optimizer for data centers and beyond. In *International Symposium on Code Generation and Optimization.* IEEE, 2–14.

Christos H. Papadimitriou and Santosh S. Vempala. 2006. On the approximability of the Traveling Salesman Problem. *Combinatorica* 26, 1 (2006), 101–120.

Robert L. Probert. 1982. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering* 8, 1 (1982), 34–42.

Vinodha Ramasamy, Paul Yuan, Dehao Chen, and Robert Hundt. 2008. Feedback-directed optimizations in GCC with estimated edge profiles from hardware event sampling. In *Proceedings of GCC Summit 2008*. 87–102.

Ching-Yen Shih, Drake Svoboda, Siao-Jie Su, and Wei-Chung Liao. 2021. Static branch prediction for LLVM using machine learning. https://drakesvoboda.com/public/StaticBranchPrediction.pdf.

Robert Endre Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

Mikkel Thorup. 1998. All structured programs have small tree width and good register allocation. *Information and Computation* 142, 2 (1998), 159–181.

László A. Végh. 2016. A strongly polynomial algorithm for a class of minimum-cost flow problems with separable convex objectives. *SIAM J. Comput.* 45, 5 (2016), 1729–1761.

Bo Wu, Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, Raul Silvera, and Graham Yiu. 2013. Simple profile rectifications go a long way. In *European Conference on Object-Oriented Programming*. Springer, 654–678.

Youfeng Wu and James R Larus. 1994. Static branch frequency and program profile analysis. In *Annual International Symposium on Microarchitecture*. ACM / IEEE Computer Society, 1–11.

Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. 2019. Can we trust profiling results? Understanding and fixing the inaccuracy in modern profilers. In *International Conference on Supercomputing*. ACM, 284–295.

Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Asia-Pacific Workshop on Systems*. ACM, 98–105.

Mingzhou Zhou, Bo Wu, Xipeng Shen, Yaoqing Gao, and Graham Yiu. 2016. Examining and reducing the influence of sampling errors on feedback-driven optimizations. *ACM Transactions on Architecture and Code Optimization* 13, 1 (2016), 1–24.