Stale Profile Matching

Amir Ayupov
Meta
Menlo Park, CA, USA
aaupov@meta.com

Maksim Panchenko
Meta
Menlo Park, CA, USA
maks@meta.com

Sergey Pupyrev
Meta
Menlo Park, CA, USA
spupyrev@meta.com

Abstract

Profile-guided optimizations rely on profile data for directing compilers to generate optimized code. To achieve the maximum performance boost, profile data needs to be collected on the same version of the binary that is being optimized. In practice however, there is typically a gap between the profile collection and the release, which makes a portion of the profile invalid for optimizations. This phenomenon is known as profile staleness, and it is a serious practical problem for data-center workloads both for compilers and binary optimizers.

In this paper we thoroughly study the staleness problem and propose the first practical solution for utilizing profiles collected on binaries built from several revisions behind the release. Our algorithm is developed and implemented in a mainstream open-source post-link optimizer, BOLT. An extensive evaluation on a variety of standalone benchmarks and production services indicates that the new method recovers up to 0.8 of the maximum BOLT benefit, even when most of the input profile data is stale and would have been discarded by the optimizer otherwise.

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

Keywords: compilers, profile-guided optimizations, profile inference, graph matching, network flow

ACM Reference Format:
Amir Ayupov, Maksim Panchenko, and Sergey Pupyrev. 2024. Stale Profile Matching. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3640537.3641573

ACM ISBN 979-8-4007-0507-6/24/03
https://doi.org/10.1145/3640537.3641573

1 Introduction

Mobile applications and ubiquitous AI workloads became an essential part of everyday life, making it crucial to optimize for their efficiency and reliability. Profile-guided optimizations (PGO), also called feedback-driven optimizations, are a collection of compiler techniques that use runtime information collected via profiling for improving the program execution. Modern PGO is successful in speeding up server workloads by providing up to a double-digit percentage boost in performance [9, 28, 38]. Similarly, PGO applied for mobile applications reduces their size and the launch time, which directly impacts user experience, and hence, user retention [10, 14, 17]. Therefore, PGO is nowadays a standard feature in most commercial and open-source compilers.

Traditionally, PGO is a combination of compiler optimizations, including function inlining, register allocation, and code layout. It relies on execution profiles of a program, such as the execution frequencies of basic blocks and function invocations, to guide compilers to optimize critical parts of the program more effectively. While many optimization passes can be applied without profile data, knowing the execution behavior of a program allows the compiler to generate a significantly optimized code. Early efforts on PGO were implemented via compile-time instrumentation, which injects code to count the execution frequencies of basic blocks, jumps, and function calls. This approach, however, not only complicates the build process, but also incurs significant performance overhead, which may alter the program's default behavior and make the collected profiles non-representative. Later works on PGO employ sampling-based approach that rely on hardware performance counters available in modern CPUs, such as Intel's Last Branch Records (LBR). Sampling-based PGO enables profiling in the production environment with a negligible runtime overhead, although the collected profiles may require an extra post-processing adjustment [9, 13, 46]. AutoFDO [3], BOLT [28], and Propeller [38] are examples of frameworks for optimizing data-center workloads based on the technology.

Continuous Profiling. While PGO systems have been successfully deployed at scale in production, there is one challenge that often remains overlooked. In order to achieve the maximum performance boost, profile data supplied to PGO needs to be statistically representative of the typical usage scenarios. Otherwise, an optimization has the potential to regress the performance instead of improving it. Consider

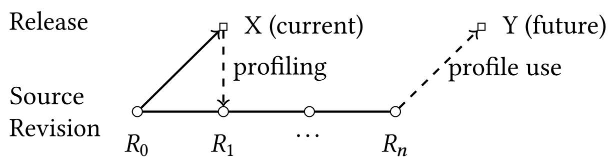


Figure 1. Continuous profiling causes a mismatch between revisions used to produce the profile (R_0) and to which the profile is applied (R_n).

for example, a function that is hot according to a profile but is rarely executed in real-world usage. A compiler may decide to inline the function, which might cause a worsened instruction cache performance due to the increased code size while not providing any runtime benefits [5, 39].

In order to collect a representative profile, continuous profiling systems are employed at large software companies [32]. These systems collect a sampled profile from the fleet and aggregate them for subsequent use in compiler optimizations. However, in such systems, the profile is always lagging the most recent source [3]; see Figure 1.

Another related issue occurs when the source code of a program is modified right before the release (which is known as a *hotfix*). Even when such changes touch a single line of the source code, they can lead to substantial changes in the generated machine code, making previously collected profiles *stale*, that is, invalid for optimizations. For these reasons, modern PGO systems assume that profiles are collected on exactly the same version of a program that is being optimized, whereas stale profile data is completely discarded.

How severe is the staleness problem in practice? One would assume that the code for large programs, referred to as *applications* or simply *binaries*, does not change too frequently and the majority of the profile data remains unchanged between consecutive releases. However, empirical data collected by the BOLT binary optimizer used for optimizing large-scale services [28, 29] disagrees with the intuition. We record 70% stale samples between two (weekly) releases for one large-scale service, and over 92% staleness after only a three-week delay of updating the profiles for another one. Thus, the benefit of applying BOLT reduces by two thirds for the former service and almost diminishes for the latter one. The phenomena are explained as follows. Profiling information is captured at the machine code level and stored at the function granularity. If a function is unchanged between two releases, the profile data can be reused for optimizations. In contrast, when the content of the function is modified, then the profile data becomes stale. Generally even innocent changes, such as adding or removing padding, result in modified jump and call addresses and their offsets from the beginning of the function, which leads to invalid profiles. In addition, function inlining results in waterfall modifications at all call sites.

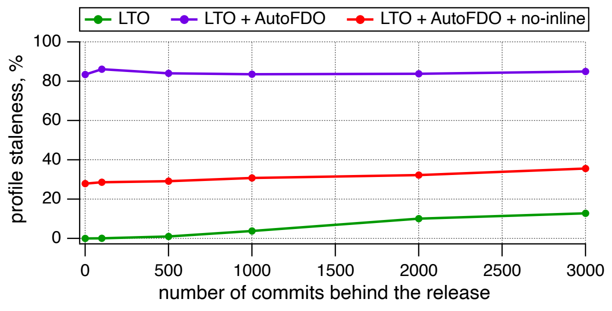
With these problems in mind, we develop a novel approach for *stale profile matching*, which is a technique for adjusting and re-using profile information gathered on a pre-release version of a program. In a sense, we relax the requirement of profiling and releasing the same version of a binary. That significantly simplifies the development process and eases the adoption of PGO systems in the real-world environment. As our key example, we show that for the large-scale `clang` binary, we recover 0.78 of PGO benefits even when over 90% of its profile data (collected on a six-month-old release) is stale. This is equivalent to a 5.9% absolute speedup of the binary on top of the state-of-the-art optimizations.

Our Contributions. The primary contributions of the work are summarized as follows.

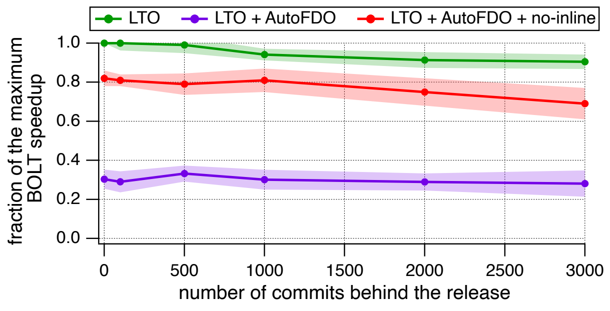
- Firstly, we thoroughly investigate profile staleness in PGO and propose a formal model for the problem, capturing practical constraints and objectives. Then, we develop a novel two-stage algorithm for the problem and demonstrate how it is applied for reducing profile staleness.
- Secondly, we present an implementation of the new approach in a post-link binary optimizer, BOLT, which is a part of LLVM [11]. The implementation is fully integrated into the main branch of the optimizer and requires no changes to the workflow for BOLT users. The algorithm is relatively simple and efficient, being able to process large production binaries without noticeable runtime overhead.
- Finally, we extensively evaluate the new approach on a variety of benchmarks, including five large open-source binaries and four production workloads at Meta. The experiments indicate that the new method recovers 0.6–0.8 of the maximum BOLT speedup, even when most of the input profile data is stale.

We emphasize that existing literature on the topic is rather sparse with only two works tackling the problem. Wang, Pierce, and McFarling [40] design a binary matching tool for stale profile propagation for Microsoft NT (2000) applications. A very recent preprint of Moreira, Pereira, and Ottoni [21] independently explore a method for using a stale profile in binary optimizers. Both methods rely on a hash-based matching between basic blocks of a program. As we argue in Section 2 and experimentally demonstrate in Section 4, such approaches are unable to provide adequate performance benefits for real-world instances. In contrast, our work presents the first two-stage algorithm comprised of *matching* and *inference* (defined in Section 3), which produces optimal and near-optimal results in practice.

We also stress that while our implementation and evaluation is done for a mainstream post-link binary optimizer, BOLT, the approach discussed in the paper is general enough to be used in other contexts. In particular, we believe that it is possible to adapt the technique to Propeller [38] or use it in conjunction with LLVM's profile-guided optimizations [3].



(a) The percentage of stale (invalid) samples in the profile collected on a binary several commits behind the release



(b) Performance regression of clang optimized with BOLT utilizing stale profile data

Figure 2. Investigation of profile staleness for the clang binary (release_15) built in different modes.

Paper Organization. The rest of the paper is organized as follows. We first investigate the phenomena of profile staleness in Section 2 and analyze the limitations of existing PGO tools. Building upon the knowledge, we develop a novel approach for re-using stale profiles. Section 3 describes the two components of our methodology, *matching* and *inference*, and describes an implementation in an open-source binary optimizer, BOLT, developed on top of LLVM. Next in Section 4, we provide a detailed experimental evaluation of the algorithm on a rich collection of open-source binaries and production workloads. Section 5 discuss related work in the area. We conclude the paper and propose possible future directions in Section 6.

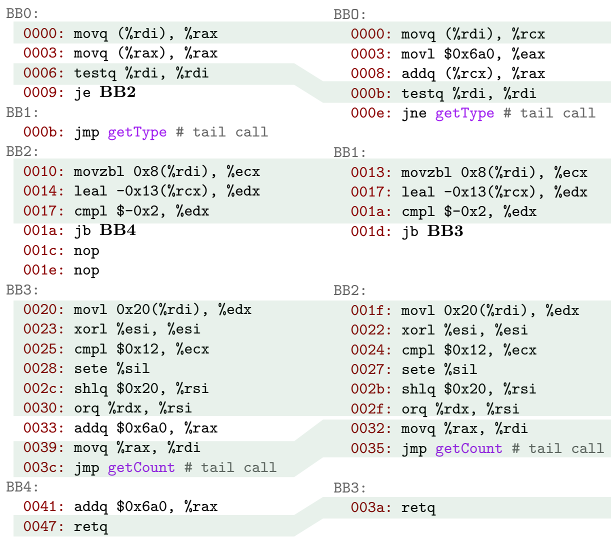
2 Investigating Profile Staleness

To shed light on the staleness problem, we investigate differences between consecutive releases of a binary optimized with BOLT. We chose to experiment with a standalone binary of the clang compiler, which has a relatively large code size, and can be easily integrated with various PGO technologies. As a baseline, we utilize release_15 of the binary cut at January 2023, and consider up to 3000 commits prior to the release so that the most stale data (3000 commits behind release_15) corresponds to June 2022.

Our first experiment, whose results are visualized in Figure 2a, measures the percentage of stale samples in the profile data collected on a binary built for the source code corresponding to x commits behind release_15. We vary x between 0 and 3000 so that $x = 0$ corresponds to the release. The level of staleness depends on how the baseline binary is built. In the “simplest” mode with the optimization level O3 and link-time optimizations (LTO) enabled, profile staleness starts at 0% for $x = 0$, that is, the profile has no stale samples. The quality of the profiles slowly degrades, as the gap between the release and the profiled binary increases. In this setup, collecting profiles on a 3-month-old binary (500 commits behind the release) invalidates less than 3% of samples, while profiling a 6-month-old binary ($x = 3000$) yields 12.8% stale samples in the profile. The results look quite different when we start utilizing compiler's PGO for building the baselines binary; in the evaluation we experiment with sampling-based AutoFDO [3]. Profile staleness reported by BOLT reaches 83% even when we rebuild clang using the same source code (the rebuild process, however, involves re-running the AutoFDO step); the value remains stable across the experiment. Such staleness almost entirely invalidates the profile data supplied to BOLT. To further understand the issue, we repeated the experiment by building the binary with inlining disabled, that is, using the O3+LTO+AutoFDO+no-inline mode. In that experiment, profile staleness is recorded at the initial value of 28% and slowly grows with the number of commits, x.

Figure 2b illustrates the performance impact of using the stale profile data. The plot reports the fraction of the maximum speedup on the clang binary achieved by BOLT utilizing stale profiles, relative to the maximum speedup it can achieve using *fresh* profile data collected on the same revision. In the evaluation, the BOLT speedup is recorded at 28% on top of the non-BOLTed counterpart in the O3+LTO mode and 12% in the O3+LTO+AutoFDO mode; the values agree with the original speedups reported by the BOLT team in [28, 29]. As expected, the impact of applying BOLT is inverse proportional to profile staleness. For the simpler O3+LTO build mode, stale profile data yields a modest 2%–3% regression. However, for the practical O3+LTO+AutoFDO mode, utilizing a 6-month-old profile results in a substantial regression: instead of the maximum possible speedup of 13%, BOLT realizes only 3.5%, which is an equivalent of value 0.26 on the plot.

In order to further understand the problem, we analyze individual functions in clang, whose profile data is marked stale. We identified three primary reasons for profile staleness. Firstly, developers modify the source code, which directly causes changes in the generated binary. Such changes include, for example, code being added or removed and function renaming or type changes. Secondly, we noticed many minor differences in the generated code, such as extra nop



```
BB0:
  0000: movq (%rdi), %rax
  0003: movq (%rax), %rax
  0006: testq %rdi, %rdi
  0009: je BB2
BB1:
  000b: jmp getType # tail call
BB2:
  0010: movzbl 0x8(%rdi), %ecx
  0014: leal -0x13(%rcx), %edx
  0017: cmpl $-0x2, %edx
  001a: jb BB4
  001c: nop
  001e: nop
BB3:
  0020: movl 0x20(%rdi), %edx
  0023: xorl %esi, %esi
  0025: cmpl $0x12, %ecx
  0028: sete %sil
  002c: shlq $0x20, %rsi
  0030: orq %rdx, %rsi
  0033: addq $0x6a0, %rax
  0039: movq %rax, %rdi
  003c: jmp getCount # tail call
BB4:
  0041: addq $0x6a0, %rax
  0047: retq
```

```
BB0:
  0000: movq (%rdi), %rcx
  0003: movl $0x6a0, %eax
  0008: addq (%rcx), %rax
  000b: testq %rdi, %rdi
  000e: jne getType # tail call
BB1:
  0013: movzbl 0x8(%rdi), %ecx
  0017: leal -0x13(%rcx), %edx
  001a: cmpl $-0x2, %edx
  001d: jb BB3
BB2:
  001f: movl 0x20(%rdi), %edx
  0022: xorl %esi, %esi
  0024: cmpl $0x12, %ecx
  0027: sete %sil
  002b: shlq $0x20, %rsi
  002f: orq %rdx, %rsi
  0032: movq %rax, %rdi
  0035: jmp getCount # tail call
BB3:
  003a: retq
```

Figure 3. An example of a function in `clang` built with AutoFDO in the profiled (left) and the release (right) binaries.

instructions or a different treatment of tail calls. Figure 3 illustrates one such example with the code of the same function in the profiled binary (left) and the release one (right). While it is easy for a human to map the basic blocks between the two versions, BOLT’s conservative strategy is to discard the function profile whenever there is a mismatch in the number of basic blocks or jumps. Such differences are a result of the existing compiler being non-stable and producing non-identical results when there are small changes in the source code or in the profile data. Finally, the above two types of modifications are often amplified by function inlining, which propagates changes in individual functions across many instances.

3 A New Approach

We assume that a *binary* (some representation of a compiled program) is a collection of *functions*. Each function is represented by a directed (possibly cyclic) control-flow graph, 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. The binary is associated with a *profile* dataset collected on a representative benchmark suite. The profile contains vertex and edge *counts*, $\text{cnt}(v) \geq 0$ for $v \in V$ and $\text{cnt}(u,v) \geq 0$ for $(u,v) \in E$, that represent the execution counts of basic blocks and jumps, respectively. As noted above, the counts might be imprecise and serve just as an estimation of actual execution counts during profiling. The profile may also contain additional meta-data, such as the names of the functions or the sizes of the basic blocks; refer to Figure 6 for an example.

We assume that there are two releases of a binary: the older one, B_{old}, and the most recent one, B_{new}. The binaries are associated with profiles P_{old} and P_{new}, respectively. We refer to Figure 4 for an example of the same function (`foo`) in the two releases of the binary along with the corresponding profiles. In the example we assume that every basic block is associated uniquely with a *hash* value, computed based on its content, that is, the opcodes and operands of its instructions. Using the hashes, it is straightforward to find a one-to-one mapping between basic blocks in the binary and in the profile, as long as the hash values stay the same, that is, when the content of the basic blocks is unchanged. However, as we argue in Section 2, that is often not the case. Our goal is to infer the counts of the basic blocks and jumps in binary B_{new} using profile P_{old}. We stress that the strategy prior our work is to discard the profile and keep unoptimized function `foo`, since its meta-data (e.g., the number of the basic blocks) does not match in B_{new} and P_{old}.

Now we describe the high-level strategy of our algorithm. As a pre-processing step independent from stale matching, BOLT finds one-to-one mapping between the functions in B_{new} and P_{old} based on their names. It first attempts to match function names exactly. For functions with unique suffixes such as produced by LLVM LTO for internal linkage symbols that may drift between compilations, BOLT attempts to match using heuristics ignoring the suffix: (i) match using a function hash, (ii) for cases where there is only one function in B_{new} and P_{old} after stripping the suffix, match them up despite hash mismatch. In the case of the remaining ambiguity, BOLT ignores the function profile. Hence, functions that have been added or deleted are discarded.

The main step of the algorithm consists of two phases, the *matching* phase and the *inference* phase. We refer to Figure 5 for an overview, which illustrates the process for function `foo` from Figure 4.

The matching phase is used to extract partial information from the profile, P_{old}, and assign *initial* counts to basic blocks and jumps in the function of B_{new}. In the example, three blocks with hashes 0xA1, 0xB2, and 0xC3 are not modified; thus, we can compute and assign their counts. Similarly, we assign initial counts for jumps whose both endpoints are matched based on the hashes. The main challenge is to define the computation of the hash values. Very strict hashes that are based on opcodes and operands of all block’s instructions are unstable to minor changes in the generated code, such as adding nops or a different register allocation. Loose hashes are more stable but might result in collisions, which is often hard to resolve correctly. Our strategy, described in detail in Section 3.1, is to define a hierarchy of hashes ranging from the strictest (based on all relevant block’s features) to the weakest (based on a few features). This strategy allows matching unchanged blocks with high confidence using the

```
BB1 [0xA1]:
  0000: testq %rdi, %rdi
  0003: je BB5
BB2 [0xB2]:
  0005: testq %rdi, %rdi
  0008: je BB6
BB3 [0xC3]:
  000a: testq %rdi, %rdi
  000d: je BB7
BB4 [0xD4]:
  000f: movq %rax, %rdi
  0012: retq
BB5 [0xE5]:
  0013: nopw %cs:(%eax,%eax)
  0020: callq log_error_x
  0025: retq
BB6 [0xF6]:
  0026: callq log_error_y
  002b: retq
BB7 [0xG7]:
  0033: callq log_error_z
  0038: retq
```

```
function_name  : foo
execution_count: 400
num_blocks     : 7
blocks:
- block_id: 1
  hash: 0xA1
  successors: {id : 2, cnt : 300}, {id : 5, cnt : 100}
- block_id: 2
  hash: 0xB2
  successors: {id : 3, cnt : 150}, {id : 6, cnt : 150}
- block_id: 3
  hash: 0xC3
  successors: {id : 4, cnt : 50}, {id : 7, cnt : 100}
- block_id: 4
  hash: 0xD4
- block_id: 5
  hash: 0xE5
- block_id: 6
  hash: 0xF6
- block_id: 7
  hash: 0xG7
```

(a) Binary B_{old} (left) with profile P_{old} (right)

```
BB1 [0xA1]:
  0000: testq %rdi, %rdi
  0003: je BB5
BB2 [0xB2]:
  0005: testq %rdi, %rdi
  0008: je BB5
BB3 [0xC3]:
  000a: testq %rdi, %rdi
  000d: je BB5
BB4 [0xXX]:
  000f: movq %rax, %rdi
  0012: setne %al
  0015: retq
BB5 [0xYY]:
  0016: callq log_error_xyz
  001b: retq
```

```
function_name  : foo
execution_count: 400
num_blocks     : 5
blocks:
- block_id: 1
  hash: 0xA1
  successors: {id : 2, cnt : 300}, {id : 5, cnt : 100}
- block_id: 2
  hash: 0xB2
  successors: {id : 3, cnt : 150}, {id : 5, cnt : 150}
- block_id: 3
  hash: 0xC3
  successors: {id : 4, cnt : 50}, {id : 5, cnt : 100}
- block_id: 4
  hash: 0xXX
- block_id: 5
  hash: 0xYY
```

(b) Binary B_{new} (left) with profile P_{new} (right)

Figure 4. An example of a function, `foo`, modified between two releases (*old* and *new*) of the binary. B_{new} and P_{old} comprise the input for the stale profile matching problem. The goal is to infer a profile, which is as close to P_{new} as possible.

strict hash and, at the same time, provide some matches for blocks with modifications. We note that a similar hash-based matching is utilized in the earlier work [40]. The difference is that we use the assigned counts as the initial guesses which can be modified at the second phase.

Once the initial counts are determined, we propagate the values through the graph and fill in the missing counts. To this end, we extend a recent work of He, Mestre, Pupyrev, Wang, and Yu [9] for profile inference. The goal of the subroutine is to guess the control flow in a graph given a partial assignment of block and jump counts. The algorithm propagates the counts along the control-flow graph to make it *consistent* and *realistic*. The former objective ensures that the sum of incoming and outgoing counts for each non-terminal vertex is the same, as in the actual program execution. The latter objective is used to distribute the counts evenly when there are multiple equally likely solutions, such as for the successors of block 0xC3 in Figure 5. Our extension of the algorithm in [9] is based of computing the maximum flow of minimum cost and described in Section 3.2.

3.1 Matching

The goal of the phase is to find matches between basic blocks within a pair of functions. This is done by calculating a 64-bit hash value for each block based on its code content. Recall that basic blocks often change between two releases of a binary; hence, a naive one-to-one comparison of the hashes will likely result in a very few matches. In order to accommodate minor changes and match as many basic blocks as possible, we introduce multiple levels of hash computation.

- A *loose* hash of a basic block is based on all distinct instruction opcodes of the block; that is, instruction operands are excluded from the computation, as they may change from one version to another. The hash is computed by creating a string from lexicographically ordered instruction opcodes, which is then hashed with a machine-independent `xxHash`. We ignore pseudo and nop instructions as well as unconditional jumps, as they are often added or removed as a result of basic block reordering.
- A *strict* hash is based on all instruction opcodes and their operands. The computation is order-dependent, that is, two blocks have the same strict hash if and only if they are comprised of the same instructions and their order is the same. Similarly to the previous case, we ignore pseudo instructions and unconditional jumps in the computation.
- A *full* hash of a basic block is based on the block's strict hash combined with loose hashes of its successors and predecessors. Empirically, the value is useful to resolve collisions between blocks with identical code content that are often a result of the function inlining pass, which substitutes calls to the same function with (identical) basic blocks of the callee.

In addition to the three values defined above, we associate each basic block with the *offset* of its address from the beginning of the function in the binary; the offset is guaranteed to be unique across basic blocks of a function. Thus, we have four hash values for every block in the binary. The values are lowered into 16-bit integers and concatenated to form a 64-bit hash value, which is stored in the profile.

In order to find a match for a basic block from P_{old} in the set of blocks corresponding to the function in B_{new}, we first check if there is a matching block with the same full hash. At this stage, the collisions (that is, blocks with the same hash value) are rare. If we find a matching candidate, we stop the matching for the block. Otherwise, we try to find a match based on the strict hash, followed by the loose hash. At these levels the probability of collisions are higher, and we break ties by choosing the candidate whose offset is the closest to

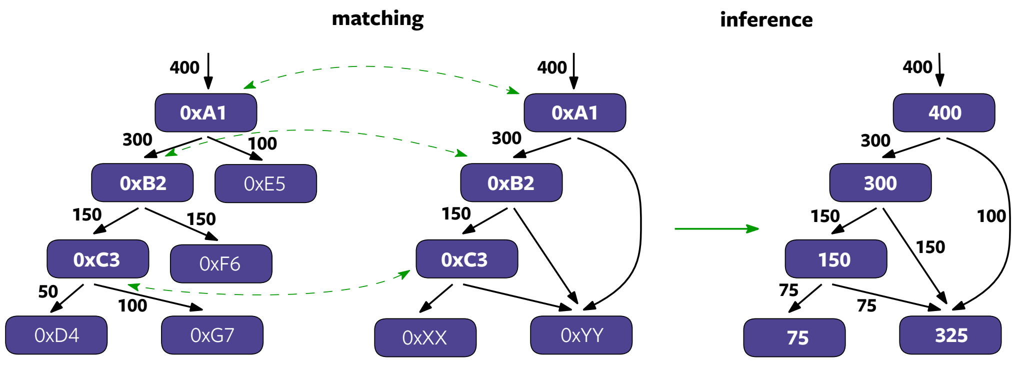


Figure 5. An overview of our algorithm for the stale profile matching problem.

the offset of the considered basic block. In addition to the rule above, we always match the first (entry) basic block from the profile to the first block in the binary. We emphasize that the hash definitions and the matching strategy is of a heuristic nature, which has been tuned on several large-scale binaries described in Section 4.1. It is likely that further fine-tuning is possible, e.g., by introducing more hash levels and including extra code features, such as function calls. In this paper, we opted for simplicity of the computation rather than potentially more accurate but less interpretable schemes based, for example, on machine learning.

3.2 Inference

The goal of the phase is to reconstruct the counts of all basic blocks and jumps, while respecting, as much as possible, the initially assigned estimations. 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, that we seek to find. Recall that some but not necessarily all vertices and edges have initial counts, denoted $\text{cnt}(\cdot)$; the case when a block or a jump is missing the initial count is indicated by $\text{cnt}(\cdot) = \varnothing$.

We want the counts to be consistent with a real execution of a function, which starts at the entry block, then traverses the blocks in some order along the edges, and ends at an exit block. Thus, we introduce the *flow conservation* rules:

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

for all non-entry non-exit vertices $v \in V \setminus T^*, v \neq s^*$, and

$$f(s^*) = \sum_{(s^*,u)\in E} f(s^*,u), \quad f(t^*) = \sum_{(u,t^*)\in E} f(u,t^*)$$

for the entry s^* and for all exits $t^* \in T^*$, respectively.

We also want to preserve the initial estimates derived at the matching phase. To measure how close the counts to the estimations, we introduce the following objective:

$$\sum_{\substack{v\in V\\ \text{cnt}(v)\neq\varnothing}} cost\big(f(v), \text{cnt}(v)\big) + \sum_{\substack{(u,v)\in E\\ \text{cnt}(u,v)\neq\varnothing}} cost\big(f(u,v), \text{cnt}(u,v)\big), \tag{1}$$

where the sum is taken over all vertices and edges with an assigned count. Here the term $cost(f, \text{cnt})$ penalizes the change of a count from f to cnt:

$$cost(f, \text{cnt}) = \begin{cases} k_{inc}(f - \text{cnt}) & \text{if } f \geq \text{cnt} \\ k_{dec}(\text{cnt} - f) & \text{if } f < \text{cnt} \end{cases}$$

Here k_{inc} and k_{dec} are non-negative penalty coefficients, whose exact values are determined in Section 4.3.

Overall, we seek to solve an optimization problem of finding consistent counts of vertices and edges of a given control-flow graph, $G = (V, E)$, with prescribed count estimations, $\text{cnt}(v), v \in V$ and $\text{cnt}(u,v), (u,v) \in E$, while minimizing objective (1). He et al. [9] show how a variant of the problem, which has $\text{cnt}(u,v) = \varnothing$ for *all* edges of G, can be mapped to an instance of the minimum-cost maximum flow problem, and hence, can be solved optimally in polynomial time. Observe that their algorithm can in addition preserve (as a secondary objective) provided branch probabilities. We can use the feature to evenly distribute the counts among the successors, when they do not have initial counts, such as for the successors of block 0xC3 in Figure 5.

Next we show how to reduce our problem to the more restricted variant studied in [9]. To this end, we identify and subdivide all edges of G that have an initial count. That is, for all $(u,v) \in E$ with $\text{cnt}(u,v) > 0$, we replace the edge by a new vertex w and two (directed) edges (u,w) and (w,v); then we set $\text{cnt}(w) = \text{cnt}(u,v)$ and $\text{cnt}(u,w) = \text{cnt}(w,v) = \varnothing$. For example, the graph in Figure 5 (middle) is modified by subdividing two edges, (0xA1, 0xB2) and (0xB2, 0xC3), and setting their initial counts to 300 and 150, respectively. Note that the new graph contains at most $|V| + |E|$ vertices and at most $2|E|$ edges, and only vertices may have initial counts. It is easy to verify that the reduction preserves the optimality of solutions. Therefore, we can apply the algorithm from [9] to solve the inference problem. We conclude that the work provides a near-linear time implementation of the algorithm in practice; we verify the runtime in Section 4.4.

```
functions:
  function_name  : foo
  function_id    : 1
  execution_count: 400
  num_blocks     : 7
  blocks:
  - block_id: 1
    execution_count: 400
    num_instructions: 20
    hash: 0xA1
    successors: [bid: 2, count: 300, bid: 5, count: 100]
  - block_id: 2
    execution_count: 300
    num_instructions: 7
    hash: 0xB2
    calls: [fid: 2, count: 300, offset: 0x30]
    successors: [bid: 3, count: 150, bid: 6, count: 150]
  - block_id: 3
    execution_count: 150
    num_instructions: 13
    hash: 0xC3
    successors: [bid: 4, count: 50, bid: 7, count: 100]
    ...
  function_name  : bar
  function_id    : 2
    ...
```

Figure 6. An example of the profile data in YAML format.

3.3 Implementation and Engineering

Here we give an overview of profiles supported by BOLT.

- **Fdata profile** is an offset-based profile, which encodes taken control-flow edges with their frequency and branch misprediction information. While the profile format is currently the default, it does not permit any discrepancies between the binary it was collected from and the binary it is applied to. Therefore, it is unsuitable for the stale matching use case.
- **Pre-aggregated perf profile** contains aggregated LBR data without binary knowledge. It encodes less information than fdata, as it lacks symbol information and omits fallthrough edges. It can be efficiently collected and stored but needs to be augmented with the binary it was collected on. This format is not self-contained and hence unsuitable for our use case.
- **YAML profile** is a structured representation, which encodes control-flow information along with function and basic block metadata. YAML profile explicitly encodes basic blocks and control-flow edges and therefore, permits some discrepancies between the profiled and the input binaries. The format is the most flexible of all three and can easily be extended with necessary basic block metadata. The primary downside is that processing YAML profile takes longer than fdata — in one representative case by about 70%. Figure 6 illustrates the YAML format.

Stale matching is implemented as a part of the BOLT `YAMLProfileReader` class. *Matching* is abstracted from BOLT IR by basic block hashes, which is added to YAML serialization format. Basic block hashing is extended to capture necessary instruction and control-flow information. *Inference* is done on an IR-independent CFG representation (`FlowFunction`), and reused in other parts of LLVM. Stale profile matching is applied immediately after attaching the profile to CFG. At this point, it is known if the function failed to match up with the profile. Therefore, the algorithm works only with function profiles that would otherwise be discarded.

4 Experimental Evaluation

To validate the effectiveness of our approach, we (i) compare the performance of binaries generated with the use of the new profile data, and (ii) evaluate the precision of the inferred block and jump frequencies. The experiments presented in this section were conducted on a Linux-based server with a dual-node 40-core 2.0 GHz Intel Xeon Gold 6138 (Skylake) having 256GB RAM, except `chromium` which was conducted on a Linux-based desktop with a 12-core 3.6 GHz Intel Core i7-12700K (Alder Lake) having 128GB RAM. The algorithms are implemented on top of `release_16` of LLVM.

4.1 Benchmarks

We evaluated our approach on large open-source applications and real-world binaries deployed at Meta's data centers; see Table 1. As publicly available benchmarks, we selected widely used programs that have large code size: two open-source compilers (`clang` and `gcc`), two database servers (`rocksdb` and `mysql`), and a browser (`chromium`). Next we discuss the testing setup for each of them.

- For `clang`, we use the `release_15` branch of LLVM as the base (B_{new}) release and the release with 3000 commits behind that for B_{old}. They are built in `O3+LTO+AutoFDO` mode, and the profiles are collected by compiling several medium-sized template-heavy C++14 source files.
- For `gcc`, we use `release-9.3.0` and `release-8.3.0` that were released approximately one year apart. We build the binaries with `O3+LTO` and collect the profiles on the same standalone benchmark of C++14 files.
- `rocksdb` is a fast key-value storage; we utilize `release-8.1.fb` and `release-7.1.fb` with ≈1000 commits from each other. The binaries are built in the `Release` mode with interprocedural optimization; `AutoFDO` is used to further speedup the binary. To collect profiles and measure performance, we run `db_bench` on three built-in benchmarks, `fillrandom`, `fillseekseq`, and `overwrite`.
- Experiments with `mysql` server are based on `mysql-8.1.0` and `mysql-8.0.28` released one year apart; we use the release version of the build using `O3+LTO` mode. We use `oltp_read_only` test from the `sysbench` benchmark suite, running 4 threads to read from the database initialized by inserting 100,000 records into 8 tables.
- For `chromium`, we use milestones 111-115, that are aligned with Chrome releases. We utilize the `official` build configuration, which makes use of bundled `clang` and enables

Table 1. Basic properties of evaluated binaries

	code size (MB)		functions		blocks per function		
	hot	all	hot	all	p50	p95	max
clang	10.4	83.2	13,444	95,943	20	274	12,464
gcc	3.6	24.1	6,971	35,442	28	278	5,974
rocksdb	0.4	5.1	1,014	9,610	13	238	1,282
mysql	1.5	23.3	2,501	46,133	6	64	5,152
chromium	18.3	171	21,244	541,722	18	151	15,856
hhvm	14	230	20,906	476,861	7	62	13,156
dc1	28.5	853	56,578	1,759,240	13	151	9,773
dc2	15.2	892	31,630	2,146,062	13	156	25,979
dc3	17.7	874	45,256	2,119,651	13	146	20,379

both LTO and PGO, with two tweaks: (i) disabled control-flow integrity, and (ii) disabled debug information generation to speed up builds. Speedometer2.0 is used to measure the performance of chromium.

Note that while the intended use of our technique is with software deployments having short release cycle, the open-source benchmarks have releases that are months apart. The aim of the experiment is to show that the approach generalizes to the large number of accumulated differences.

For data-center workloads, we selected four binaries. The first system is the HipHop Virtual Machine (hhvm), that serves as an execution engine for PHP at Meta, Wikipedia, and other large websites. The other three binaries are large application running inside Meta's data centers. We use two consecutive releases for each of the binaries, which are typically done on a weekly and bi-weekly basis. These binaries are built with clang in O3+LTO mode and use the compiler's PGO to enhance their performance.

4.2 Binary Performance

Figure 7 presents an evaluation on the open-source benchmarks. Here we compare the following alternatives:

- the original non-BOLTed binary created by the compiler; this is our *baseline* for the comparison;
- the existing strategy in BOLT that uses *stale profile* discarding all functions with invalid profiles;
- an optimization using the most recent *fresh profile* data; note that the option is not practical and used only as an upper bound for optimizations;
- the newly proposed *stale profile matching* approach.

For each benchmark, the figure shows the absolute speedup of the three strategies on top of *baseline*, indicated by the number on top of the bars. The height of each bar is proportional to the fraction of the maximum possible performance improvement achieved by each strategy. The values are obtained by repeating each experiment 100 times to increase the precision of the measurements so that the average mean deviation is within 0.05%; thus, we omit the deviations from the figure. We observe that *stale profile matching* is able to recover 0.64–0.79 of the maximum possible speedup achieved by BOLT using fresh profile data. This is equivalent to 6.7% (for mysql) and 5.9% (for clang) absolute performance boosts on the benchmark. In contrast, the existing strategy recovers only 0.15–0.45 of the maximum speedup.

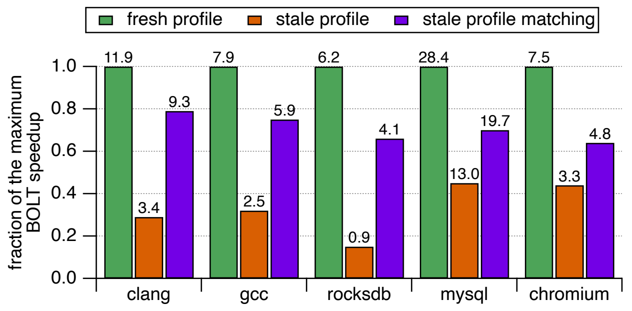


Figure 7. Relative and absolute performance improvements on the open-source benchmarks. The heights of the bars correspond to the fraction of the maximum performance improvement for each of the modes, while the values on top indicate absolute speedups.

To better understand the benefits of applying the new algorithm, we collect several perf metrics related to code layout. As expected, the main advantage of the algorithm is an improved performance of the L1 instruction and I-TLB caches; refer to Figure 8 where we use frontend_retired.l1i_miss and frontend_retired.itlb_miss hardware events for estimating I-cache and I-TLB misses, respectively. We also see a modest improvement in the number of branch misses and the performance of the last level cache, though the absolute difference is less prominent.

In order to measure the impact of stale profile matching on production workloads, we use internal performance measurement tools for running A/B experiments at Meta. We stress that the obtained measurements on the workloads are noticeably noisier than on the standalone open-source benchmarks; empirically typical deviations are up to 0.4% for dc binaries and up to 0.2% for hhvm. For hhvm, we record a 5.7%±0.2% performance improvement by using *fresh profile*, which degrades to a 3.1% value for *stale profile* containing 64% of stale samples. The new *stale profile matching* achieves 4.5% performance boost on top of *baseline*, which is equivalent to recovering 0.79 of the maximum speedup. For dc services, the existing experimentation infrastructure only allows to measure absolute speedups of the new approach over *stale profile*. We record a 0.2%±0.4% boost for dc1 containing 14% stale samples, 0.5%±0.3% boost for dc2 containing 21% stale samples, and 0.6%±0.3% boost for dc3 containing 29% stale samples. That is, for three out of four production workloads, we are able to measure a statistically significant improvement by turning on the technique.

Finally, we discuss the performance of *stale profile matching* with respect to alternative solutions. A recent work of Moreira et al. [21] provides an evaluation of their technique,

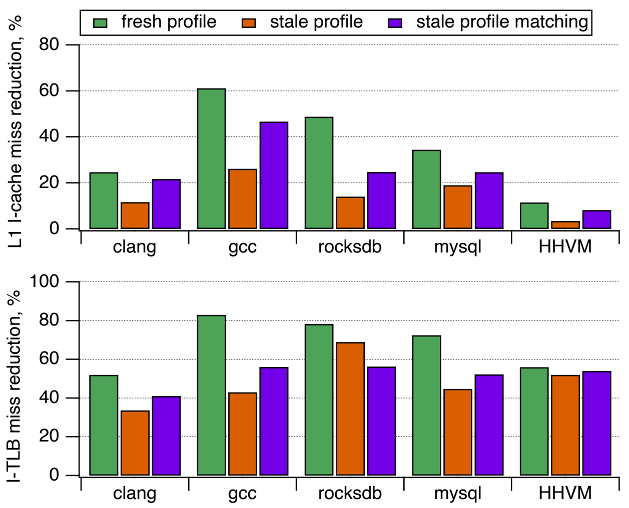


Figure 8. Improvements of the instruction cache `perf` metrics on top of non-BOLTed binaries.

referred to as Beetle, as well as the earlier approach, BMAT, by Wang et al. [40]. In the evaluation, Moreira et al. [21] estimates the impact of Beetle and BMAT on a number of open-source binaries. We observe that their evaluation differs from our setup described in Section 4.1 in that it does not employ a compiler's PGO, and hence, a direct comparison of the results is impossible. To mimic the setup of [21], we repeat the experiment with the `clang` binary by building its `release_14` with `O3+LTO`, while using `release_10` for collecting profile data. Similar to [21], we record 75% of stale functions in the profile, and the speedups achieved by *fresh profile* and *stale profile* are 28% and 6%, respectively; all three measurements closely match the reported values in the paper. Moreira et al. [21] provide evaluations of multiple techniques on the benchmark, including Beetle and BMAT, and none of them exceeds a 7% speedup on top of non-BOLTed binary. In contrast, *stale profile matching* yields a substantially higher 18% performance boost. Analogously, [21] reports the mean recovered fraction of the maximum BOLT speedup on several benchmarks to be 0.25 for BMAT and 0.43 for Beetle, while *stale profile matching* recovers 0.6–0.8 of the speedup on the arguably more challenging `O3+LTO+AutoFDO` setup. Therefore, we do not consider earlier studies in [21, 40] as viable alternatives for the problem.

4.3 Quality of Inferred Profiles

To evaluate the accuracy of the algorithm, we compare the resulting profile data of with *fresh profile*. To this end, we employ two measures that quantify the similarity between the two profiles. The first one follows earlier works on profile inference and referred to as the *edge overlap* [4, 9]. Let $f(e), e \in E$ be the constructed edge counts by the algorithm, and $gt(e), e \in E$ be the counts of the edges in the fresh profile.

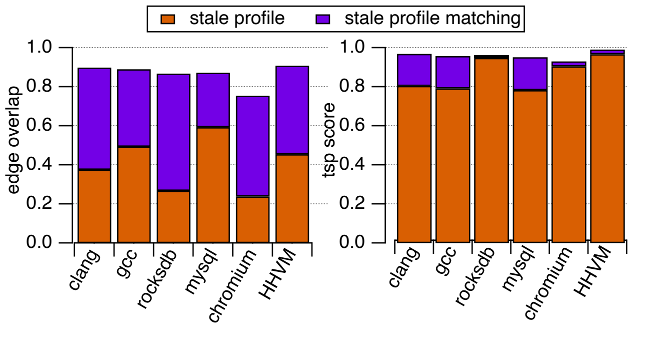


Figure 9. Edge overlap and tsp score for the benchmarks.

The *edge overlap* is defined as

$$\sum_{e \in E} \min\left(\frac{f(e)}{\sum_{e \in E} f(e)}, \frac{gt(e)}{\sum_{e \in E} gt(e)}\right),$$

where the sum is taken over all edges of the graph. If the counts in the two profiles match exactly, the overlap is equal to 1; otherwise, the measure takes values between 0 and 1. The second measure is called the *tsp score*. It is motivated by studies on code layout [20, 23, 30] in which co-locating basic blocks that frequently call each other is beneficial for the performance. To compute the value, we apply basic block reordering for all functions in a binary based on the fresh profile, and calculate the sum of counts for all *fallthrough* edges, that is, edges between pairs of consecutive blocks; denote the value by $\text{TSP}(gt)$. Next, we compute $\text{TSP}(f)$, where we use inferred counts, f, for basic block reordering but fresh counts for the summation. Observe that $\text{TSP}(gt) \geq \text{TSP}(f)$ for the majority of instances since existing algorithms for basic block reordering produce near-optimal layouts with respect to the measure [20, 23]. The *tsp score* is defined by $0 \leq \frac{\text{TSP}(f)}{\text{TSP}(gt)} \leq 1$ and estimates how suitable are the constructed profiles for code layout.

Figure 9 demonstrates the analysis of the two measures on open-source benchmarks and `hhvm`. Edge overlap values range from 0.76 (for `chromium`) to 0.91 (for `clang`, `gcc`, and `hhvm`), which is a substantial improvement over 0.24–0.6 for the stale data. For the *tsp score*, we record an improvement from 0.8–0.9 to 0.93–0.96 on the open-source benchmarks and to the value of 0.99 on `hhvm`. Loosely speaking, the latter result indicates that in 99% of instances, the generated profile is as good for basic block reordering as the fresh one. We summarize the findings by noting that *stale profile matching* is able to significantly reduce but not completely eliminate the gap to the fresh profile data.

The primary parameters of *stale profile matching* are penalty coefficients, k_{inc} and k_{dec}, introduced in Section 3.2. Using the *edge overlap* score, we identified the following combination resulting in the highest value: $k_{inc} = 1$ and $k_{dec} = 2$. Intuitively, it is twice more expensive to decrease a block count from its initial estimate than to increase it.

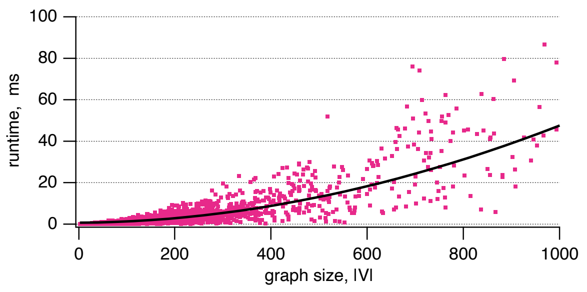


Figure 10. The runtime (in milliseconds) of *profi* as a function of the number of basic blocks measured on hhvm.

4.4 Build Time

Here we evaluate the runtime of *stale profile matching* and its impact on the overall BOLT's processing time. The absolute running times (in milliseconds) of the algorithm are illustrated in Figure 10. While the observed complexity of the algorithm is super-linear in the number of basic blocks, the runtime does not exceed 0.1 second even for instances containing $|V| = 1000$ blocks. The majority of real-world instances contain much fewer vertices (see Table 1), and the largest recorded runtime across all functions in our dataset is 0.8 second. Hence, our algorithm is unlikely to introduce an overhead to the existing build process of data-center applications. The total time taken by *stale profile matching* for 21K functions of hhvm is under 20 seconds, which is only a small fraction of the overall processing time for the binary [29].

5 Related Work

There exists a rich literature on profile-guided compiler optimizations. Traditionally profile data is used for function inlining and outlining [6, 15], basic block reordering [12, 23, 30], function layout [10, 27, 30], merging similar functions [34, 35], loop optimization [33], register allocation [7, 18], and many others. These techniques are implemented in a variety of compilers and binary optimizers used for static and dynamic languages [25, 26], and can be applied at the compile time [3], link time [17, 37], or post-link time [19, 28, 38].

Most of the above tools assume that a representative profile dataset is available. In practice, however, it is often difficult to generate an adequate profile even for a binary processing a specific benchmark. Chen et al. [4] observe hardware-related problems that lead to inaccurate profiles and suggest several tricks to mitigate the hardware bias. Later studies explore alternative profile improvement techniques, such as varying the sampling rate [41] and utilizing LBR technology available on modern Intel x86 processors [3, 24]. However, not all processor vendors provide such logging functionality, and moreover, recent studies indicate that LBRs still suffer from sampling bias [43, 44]. Therefore, it is common to apply a post-processing step to correct the profiles[9, 16]. An alternative direction to improve profile accuracy is to employ machine learning to predict the frequencies of the blocks, functions, and branch probabilities [2, 42]. Despite a wealth of recent studies in the area [22, 31, 36], we are not aware of a successful application of such a technique at scale in production environment.

The problem explored in this paper is complementary to the above studies and has been mostly overlooked by the community. In order to reuse profile data collected on previous releases of an application, the BMAT system by Wang et al. [40] uses a hash-based matching between basic blocks. Their approach is equivalent to the first phase of our algorithm; this phase alone, as we show in the experimental evaluation, is not competitive with the two-phase approach. Another closely related (and still unpublished) work is the Beetle technique by Moreira et al. [21]. The authors suggest to extract certain branch characteristics (e.g., direction and opcode) and use them to map profile information across releases. Similarly to our study, the work [21] is evaluated in BOLT. Section 4 indicates that our approach achieves substantial performance gains over Beetle. Finally, we mention another recent stream of work that might mitigate the profile staleness: an online (or just-in-time) system for optimizing binaries written in unmanaged languages. In that scenario, the binary starts running in an unoptimized state and its profile is collected during a certain warm-up period. Once enough profile data is collected, we apply optimizations and shift the execution to the new code. Ocolos is one recent prototype implementing such an approach [45]. While the profile staleness problem is eliminated, this approach introduces significant profiling and optimization overhead that could be amortized for long-running binaries.

More generally, the problem of inferring stale profiles can be seen as a special variant of binary code similarity, whose goal is to identify differences and similarities between two pieces of binary code. This is fundamental task for many applications; refer to the recent surveys on the topic [1, 8].

6 Conclusion

We designed and implemented a novel approach for re-using profile data collected on binaries built from several revisions behind the release. Based on an extensive evaluation, we conclude that the proposed technique can recover a large portion of the performance loss due to outdated profiles. One direct implication is a simplified maintenance of performance-critical applications that are frequently released; by introducing stale profile matching, such applications can be released without re-collecting profiles after every hotfix. A possible future direction is to adapt our implementation in BOLT to other PGO tools, such as Propeller [38] or AutoFDO [3]. While we do not foresee any high-level obstacles, there might be non-trivial implementation challenges.

References

[1] Saed Alrabaee, Mourad Debbabi, Paria Shirani, Lingyu Wang, Amr Youssef, Ashkan Rahimian, Lina Nouh, Djedjiga Mouheb, He Huang, Aiman Hanna, et al. 2020. Binary analysis overview. *Binary Code Fingerprinting for Cybersecurity: Application to Malicious Code Fingerprinting* (2020), 7–44. https://doi.org/10.1007/978-3-030-34238-8_2

[2] Thomas Ball and James R Larus. 1993. Branch prediction for free. In *Programming Language Design and Implementation (PLDI)*, Robert Cartwright (Ed.), Vol. 28. ACM, 300–313. https://doi.org/10.1145/155090.155119

[3] 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, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[4] 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. https://doi.org/10.1109/TC.2011.233

[5] William Y Chen, Pohua P. Chang, Thomas M Conte, and Wen-mei W. Hwu. 1993. The effect of code expanding optimizations on instruction cache design. *Transactions on Computers* 42, 9 (1993), 1045–1057. https://doi.org/10.1109/12.241594

[6] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for code size reduction. In *25th Brazilian Symposium on Programming Languages*. ACM, Joinville, Brazil, 17–24. https://doi.org/10.1145/3475061.3475081

[7] Andreas Fried, Maximilian Stemmer-Grabow, and Julian Wachter. 2023. Register allocation for compressed ISAs in LLVM. In *International Conference on Compiler Construction* (Montréal, QC, Canada). ACM, New York, NY, USA, 122—-132. https://doi.org/10.1145/3578360.3580261

[8] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM Comput. Surv.* 54, 3 (2021), 51:1–51:38. https://doi.org/10.1145/3446371

[9] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile inference revisited. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–24. https://doi.org/10.1145/3498714

[10] Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing function layout for mobile applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, NY, USA, 52–63. https://doi.org/10.1145/3589610.3596277

[11] 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. https://doi.org/10.1109/CGO.2004.1281665

[12] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *International Conference on Compiler Construction*. ACM, Washington, DC, USA, 65–75. https://doi.org/10.1145/3302516.3307358

[13] Byeongcheol Lee. 2015. Adaptive correction of sampling bias in dynamic call graphs. *ACM Transactions on Architecture and Code Optimization* 12, 4 (2015), 1–24. https://doi.org/10.1145/2840806

[14] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient profile-guided size optimization for native mobile applications. In *International Conference on Compiler Construction*. ACM, Seoul, South Korea, 243–253. https://doi.org/10.1145/3497776.3517764

[15] Kyungwoo Lee, Manman Ren, and Shane Nay. 2022. Scalable size inliner for mobile applications (WIP). In *International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, San Diego, CA, USA, 116–120. https://doi.org/10.1145/3519941.3535074

[16] 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. https://doi.org/10.1007/978-3-540-77560-7_20

[17] Gai Liu, Umar Farooq, Chengyan Zhao, Xia Liu, and Nian Sun. 2023. Linker code size optimization for native mobile applications. In *International Conference on Compiler Construction*. ACM, New York, NY, USA, 168–179. https://doi.org/10.1145/3578360.3580256

[18] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial register allocation and instruction scheduling. *Transactions on Programming Languages and Systems* 41, 3 (2019), 1–53. https://doi.org/10.1145/3332373

[19] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: A post-link optimizer for the Intel® Itanium® architecture. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, IEEE / ACM, San Jose, CA, USA, 15–26. https://doi.org/10.1109/CGO.2004.1281660

[20] Julián Mestre, Sergey Pupyrev, and Seeun William Umboh. 2021. On the Extended TSP problem. In *International Symposium on Algorithms and Computation (LIPIcs, Vol. 212)*, Hee-Kap Ahn and Kunihiko Sadakane (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:14. https://doi.org/10.4230/LIPICS.ISAAC.2021.42

[21] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Pereira. 2023. Beetle: A feature-based approach to reduce staleness in profile data. *Authorea Preprints* (2023).

[22] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485521

[23] Andy Newell and Sergey Pupyrev. 2020. Improved basic block reordering. *IEEE Trans. Comput.* 69, 12 (2020), 1784–1794. https://doi.org/10.1109/TC.2020.2982888

[24] 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.

[25] 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. https://doi.org/10.1145/3192366.3192374

[26] 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, Seoul, Korea (South), 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

[27] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *International Symposium on Code Generation and Optimization*. IEEE, Austin, USA, 233–244. https://doi.org/10.1109/CGO.2017.7863743

[28] 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, Los Alamitos, CA, USA, 2–14. https://doi.org/10.1109/CGO.2019.8661201

[29] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. 2021. Lightning BOLT: powerful, fast, and scalable binary optimization. In *International Conference on Compiler Construction*. ACM, New York, NY, USA, 119–130. https://doi.org/10.1145/3446804.3446843

[30] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. In *Programming Language Design and Implementation (PLDI)*, Vol. 25. ACM, 16–27. https://doi.org/10.1145/93542.93550

[31] Easwaran Raman and Xinliang David Li. 2022. Learning branch probabilities in compiler from datacenter workloads. *arXiv preprint arXiv:2202.06728* (2022).

[32] 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 (2010), 65–79.

https://doi.org/10.1109/MM.2010.68

[33] Rodrigo CO Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop rolling for code size reduction. In *International Symposium on Code Generation and Optimization*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, Seoul, Republic of Korea, 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[34] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function merging for free. In *International Conference on Languages, Compilers, and Tools for Embedded Systems*, Jörg Henkel and Xu Liu (Eds.). ACM, Virtual Event, Canada, 110–121. https://doi.org/10.1145/3461648.3463852

[35] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective function merging in the SSA form. In *International Conference on Programming Language Design and Implementation*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, London, UK, 854–868. https://doi.org/10.1145/3385412.3386030

[36] Nadav Rotem and Chris Cummins. 2021. Profile guided optimization without profiles: A machine learning approach. *arXiv preprint arXiv:2112.14679* (2021).

[37] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on Binary Rewriting*. 1–7.

[38] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Vancouver, BC, Canada, 617–631. https://doi.org/10.1145/3575693.3575727

[39] Theodoros Theodoridis, Tobias Grosser, and Zhendong Su. 2022. Understanding and exploiting optimal function inlining. ACM, New York, NY, USA, 977—-989. https://doi.org/10.1145/3503222.3507744

[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] 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, Berlin, Heidelberg, 654–678. https://doi.org/10.1007/978-3-642-39038-8_27

[42] 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. https://doi.org/10.1145/192724.192725

[43] 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, New York, NY, USA, 284–295. https://doi.org/10.1145/3330345.3330371

[44] 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. https://doi.org/10.1145/3409963.3410490

[45] Yuxuan Zhang, Tanvir Ahmed Khan, Gilles Pokam, Baris Kasikci, Heiner Litz, and Joseph Devietti. 2023. Online code layout optimizations via OCOLOS. *IEEE Micro* 43, 4 (2023), 71–79. https://doi.org/10.1109/MM.2023.3274758

[46] 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. https://doi.org/10.1145/2851502

Received 09-NOV-2023; accepted 2023-12-23