Are We There Yet? Unraveling the State-of-the-Art Binary Feedback-directed Optimizations

Mingliang Liu Shanlin Deng Baojian Hua*
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{liumingliang, dengshanlin}@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Binary feedback-directed optimization (BFDO) is a novel technique in optimizing program binaries leveraging dynamic profiles, and has shown promising potentials in scenarios such as data centers or clouds. However, state-of-the-art BFDOs only focused on specific instruction set architectures as well as programming languages, but neglected the architecture discrepancies, language deversities, and optimization options. As a result, a comprehensive study and understanding of state-of-the-art BFDO is still lacking, hindering its potential applications.**

In this paper, to fill this knowledge gap, we present the first systematic study of BFDO to gain an undersanding of its current capabilities, remaining grand challenges, and future research opportunities. Specifically, we first conducted an empirical study with a novel end-to-end evaluation tool we developed, to investigate current capabilities of BFDO. We identified three root causes leading to BFDO failures, stemming from binary disassembly and symbol relocation. We also revealed five factors affecting BFDO effectiveness, from compiler support code, biased profile, symbol relocation and more. Based on these findings and insights, we present four best practices for improving BFDO and propose three potential research opportunities, shedding light on its future research directions. Applying our best practices to benchmarks on RISC-V, we improve the performance by 4.57% on average, and avoided one relocation failure.

***Index Terms*—Compiler optimizations, binary feedback-directed optimizations, profile-guide optimizations**

I. Introduction

Binary feedback-directed optimization (BFDO) [1] [2] [3] [4] [5] is a post-link compiler optimization to optimize a target program's binary directly, by leveraging the dynamically collected runtime profiles. As BFDOs can map runtime profiles more precisely to binaries [6] than to high-level program sources or compiler intermediate representations, they can improve program performance beyond what static compilers or tranditional profile-gudied optizations (PGOs) [7] [8] [9] [10] can typically achieve. For example, BOLT [1], a recent proposed BFDO for x86-64, achieved up to 7.0% speedups for data-center applications. In the coming decade of data centers and cloud computing, a desire to to further improve the program efficiency with lower energy comsumption will make BFDO a promising technique to depoly.

While exiting BFDOs have shown promosing potentials in achieving considerable performance improvements for practical workloads, they still have many limitations and a thorough understanding of the state-of-the-art BFDOs is still lacking. In particular, as BFDOs optimize program binaries compiled from diverse programming languages for different instruction set architectures, with flexible compiler optimization combinations, we argue that an effective and end-to-end BFDO should fulfill the following three requirements:

- **R1-Diverse languages support**: it should effectively optimize binaries compiled from diverse high-level languages with different programming paradigms.
- **R2-Multiple architectures**: it should achieve consistent optimzition results on different instruction set architectures.
- **R3-Compiler optimization neutrality**: it should be netrual to any compiler optimizations used to generate the target binaries.

TABLE I: An overview of existing research work on BFDOs.

Research	Arch	Lang&Proj	Neutrality	Tests
BOLT [1]	x86-64	1/8	✗	8
Lightning BOLT [2]	x86-64	1/7	✗	7
Propeller [3]	x86-64	1/7	✗	7
Ispike [4]	Itanium	1/12	✗	12
CodeMason [5]	x86-64	1/1	✗	1
Our work	Three[1]	3/16	✓	48

[1] x86-64, AArch64, and RISC-V.

Unfortunately, as shown in Table I, none of the existing studies and systems satisfy these requirements. First, existing studies primarily focused on specific languages such as C/C++ thus failed to fulfill **R1**. Practically, while C/C++ are widely adopted languages for programming cloud and data centers [11] [12] [13], other languages (e.g., Go or Rust) with different programming paradigms are also rapidly deployed. For example, languages such as Go and Rust are gaining rapid adoptions [14], and their rankings are constantly improving. Go has powered numerous projects in Google and has spawned excellent open-source projects such as Docker and Kubernetes [15], while Rust are famous for its strong security guarantees, such as in Firefox [16] and Linux kernel [17] [18]. Meanwhile, binaries compiled from those languages manifest different characteristics compared with those compiled from C/C++, due to differenes in compiling techniques as well as runtime organizations. For example, binaries compiled from functional programming languages such as ML typically consist of more smaller basic blocks due to the use of continuations [**?**] [**?**] [**?**

]. As another example, binaries compiled from Go programs contains richer runtime information such as garbage collectors [19] which are absent from C/C++ binaries. As a result, the effectivness of BFDO on those diverse languages remains unknown.

Second, existing studies [1] [2] [3] [5] only focused on single architecture (specifically, x86-64), thus do not fulfill **R2**. We specualte the reason is not only x86-64's wide adoptions in data centers or clouds, but also its hardware functionalities such as Last Branch Record (LBR) which is crucial to generate precise runtime profiles. However, as other architectures such as AArch64 and RISC-V are also increasely deployed, we argue that BFDO's advantages on x86-64 does not necessarily hold for these RISCs, for several technical reasons. First, the required static disassembly required for BFDOs is a well-known unsolved problem [20], and is particularly challenging for RISC architectures, due to the common mixing of data and instructions[21]. Second, profiling hardware on RISCs have dramatically different capabilities [22] [23], hence, profiles produced by these hardware might not be precise enough to generate effective BFDO results. Unfortunately, an extensive understanding of BFDOs on these architectures is still lacking.

Third, existing studies [1] [2] [3] [4] [5] have largely ignored impacts of compiler optimizations leveraged to produce the target binaries, thus failed to fulfill **R3**. As compiler optimizations are performed at an early stage in the compiling pipeline yet do not generally come with a forward design for the generated binaries, hence they might make state-of-the-art BFDOs ineffective. For example, the default optimization level (`O0`) might generate more smaller basic blocks, posing challenges for sampling-based profile generations . As another example, linker relaxation optimizations [24] substitutes a general call instruction (e.g., `jalr`) with a shorter one with narrow ranges (e.g. `jal`), hindering the function reordering a BFDO can perform.

In light of these limitations, in this paper, we present the first comprehensive study of state-of-the-art BFDOs to investigate their current capabilities, remaining grand challenges, and future research opportunities, shedding light on future research directions. Specifically, we first designed a novel automated and scalable tool BINSIGHT to aid in our investigation. Using BINSIGHT, we investigate state-of-the-art BFDOs with both quantitative and qualitative methodologies. We first obtain quantitative results by executing BINSIGHT on micro-benchmarks as well as realworld workloads. We then performed a qualitative study with these results to investigate root causes, limitations, best practices, and opportunities. In particular, our investigation fulfills the three aforementioned requirements. **R1**: we study binaries compiled from four programming languages: C/C++, Go, ML, and Rust. We select these four language languages because they not only have different compiler toolchains hence different binary layouts, but also represent diverse programming paradigms (i.e., imperative, object-oriented, and functional programming). **R2**: we study BFDOs on three representative architectures: x86-64, AArch64, and RISC-V. We select these architecture as they not only represent different application scenarios including server, embedded, and mobiles, but also have different dynamic profiling capabilities which are crucial for BFDOs. **R3**: we experiment with not only different compile-time optimization options, but also link-time optimizations such as linker relaxation. Nevertheless, thanks to the scalability design of BINSIGHT, our system can also be used to study other potential combination of languages, architectures, and optimization options (as discussed in § III).

We obtained several findings and insights from our study. First, we identified three major root causes leading to BFDO failures, stemming from disassembly or symbol relocation; Second, we revealed five factors that significantly impact effectiveness of BFDOs, including compiler support code, biased profiles, failing branch prediction, among others. Based on our findings, we proposed four best practices for using BFDO, and point out three potential directions for future studies. By applying our best practices to benchmarks on RISC-V, we improved performance by 4.57% on average than state-of-the-art, meanwhile avoided one relocation failure.

To the best of our knowledge, this work is the *first* comprehensive study to understand state-of-the-art BFDOs. In summary, our work makes the following contributions:

- **Comprehensive study.** We systematicallly studied state-of-the-art BFDOs with quantitative and qualitative methodologies.
- **Findings and insights.** We identified three root causes leading to failures, obtained useful findings and insights.
- **Best practices and suggestions.** We proposed four best practices for using BFDO, and identify three future research directions.
- **Open source.** We open source our software prototype, benchmarks, test scripts, and evaluation results, in the interest of open science.

The rest of this paper is organized as follows: Section II presents background for this work. Section III presents the design of our evaluation approach. Section IV presents the evaluations we conducted, as well as the root cause analysis we performed. Section V proposes best practices and presents potential research opportunities. Section VI and VII discusses limitations and related work, respectively. Section VIII concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work: Profile-guided optimization (PGO) (§ II-A), Post link optimization (§ II-B), and Profile generation (§ II-C).

A. Profile Guided Optimization (PGO)

History. Profile guided optimization (PGO), also known as feedback-driven optimization (FDO), has been well studied with a long history for the potential to leverage runtime profile in guiding optimization. Studies of PGO back at least to 1960s [25, 26, 27]. Recently, with better hardware support and more accurate profile utilization, PGO has been extensively

studied and there comes out many variants, *e.g.* AutoFDO[7], SamplingPGO[8], FSAutoFDO[9], CSSPGO[10].

Feature. PGO employs a two stage build as shown in Figure 1. In the first stage, a binary is executed using a set of inputs for training and profile data is collected during the process (②). Generally, two common approaches are used to obtain program profiles for PGO: instrumentation-based and sampling-based. For instrumentation-based PGO, the compiler instruments the target program with instrumentation code. The instrumented code and program will be compiled into instrumented binaries, and then the instrumented binaries will be executed to collect important runtime information such as execution frequency of basic block, function or loop. For sampling-based PGO, source files go through the compilation pipeline as normal (①), and hardware support is used to sample the value of the instruction pointer to approximate execution frequencies. With the profile collected in stage one, the second build step compiles the binary again (④) leveraging the profile (③). For instance, the compiler may make more accurate inline decisions for frequently called functions, thereby improving program performance.

Application. As a promising compiler optimization, PGO has been widely deployed. Mainstream compilers, *e.g.* GCC[28] and LLVM[29], have complete support of PGO. Apart from the support of compilers, PGO has been successfully used in optimizing a large spectrum of real world programs, *e.g.* Chrome[7], PHP[30], .NET[31] and even Linux kernels[32]. PGO is also widely used to optimzie data-center applications. At Meta, most compute-bound services are optimized with sampling-based PGO[10].

B. Binary Feedback-directed Optimization

History. Since binary feedback-directed optimization (BFDO) is easy to use profile feedback and is applicable when source files are unavailable, this technique has been studied extensively in the past [4, 33, 34], and recent study[1] presented a novel binary feedback-directed optimizer called BOLT, demonstrating injecting profile data later may enable more accurate use of the information for better code layout, showing the considerable protential of BFDO. Inspired by BOLT, more research of BFDO optimizer carried out, such as a post link PGO tool called HALO[35], an improved version of the BOLT called Lighting BOLT[2], a new BFDO approach without disassembly called Propeller[3].

Feature. The workflow of BFDO can also be concluded in two stages. A profile is generated at the first stage via instrumentation-based or sampling-based profiling as mentioned in Section II-A. In the second stage, BFDO operates directly on the executable file (⑤) with the profile data from the first stage. Since optimizers work at the binary level, they have a global view of the program, enabling more aggressive optimizations. Optimizations, such as code layout, data compaction, hot/cold code splitting can be introduced by rewriting the executable.

Application. The deployment of BFDO is constantly promoted. LLVM has added it as a component[36]. And BFDO has been successfully used in optimizing GCC[1], Clang[37], Linux kernel[38]. At present, BFDO is also developing its use in business, as it has been deployed to production at Google[3] and about 1/7 compute-bound services running at Meta data centers are optimized by it[10].

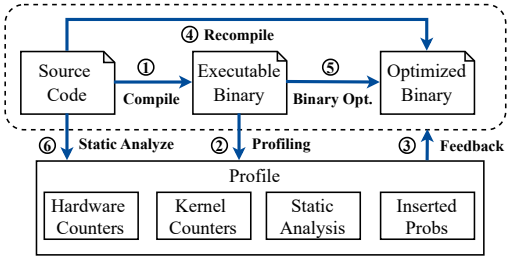


Fig. 1: Workflow of PGO and BFDO.

C. Profile Generation

In general, four profile generation approaches are commonly used, three of which are dynamic methods called instrumentation-based, sampling-based and hardware-based, and a novel static method.

Instrumentation-based. Instrumentation-based profile generation, is a technique used to track accurate execution counts for various paths of a program during runtime by inserting instructions into the code. Since instrumentation-based profiling provides precise data, it also introduces non-trivial runtime overhead, and instrumented binaries usually cannot be run in production environments, which hinders the use of this method. Notwithstanding these limitations, instrumentation-based profiling has made considerable progress and there comes out many instrumentation tools, such as Atom[39], Valgrind[40], and Pin[41].

Sampling-based. Sampling-based profile generation periodically interrupts programs, and collects snapshots of their running state as profiles with negligible runtime overhead. Traditional sample-based profiling is a statistical estimate which may produce inaccurate profile. To mitigate this problem, continuous profiling[42, 43] was proposed and hardware-based profiling[6, 44, 45] was introduced as an significant enhancement of sampling-based profiling.

Hardware-based. Hardware-based profiling[44] is an effective method for collecting profile information using hardware performance counters. These hardware critically rely on the architectures. For example, Intel x86-64 equip the hardware functionality called Last Branch Records (LBR)[46]; AArch64 also has similar hardware called Embedded Trace Macrocell (ETM)[22]; and RISC-V has Hardware Performance Monitor (HPM) which can also be used to profile different events[47, 48].

Static profiling. Without access to inputs of a program, a static profiler only needs the program's code to estimate the chance that a conditional branch can be taken. In the 1990s, Calder et al.[49] presented a method, called Evidence-Based

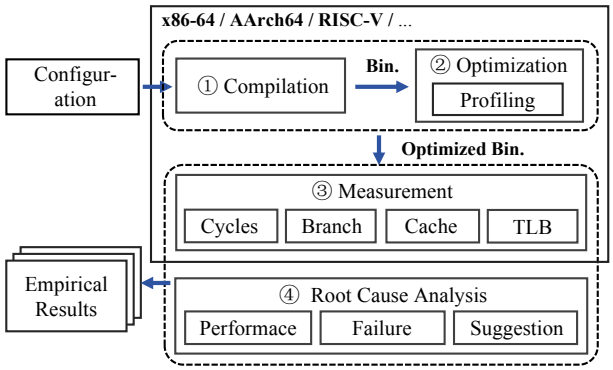


Fig. 2: BINSIGHT Architecture.

Static Prediction (ESP), to use machine-learning techniques to predict branch outcomes statically. A recent study[50] revisited the static profiling technique, proposed an adaptation called Vintage ESP Amended (VESPA), and demonstrated the profile it generated can bring considerable performance improvement on top of highly optimized binaries.

III. APPROACH

This section presents our approach to conducting the evaluation process. It is challenging to perform an evaluation of various datasets on multiple platforms, for two key reasons: 1) automation and 2) scalability. First, the evaluation should be fully automated, otherwise it is difficult and time-consuming to evaluate a mass of projects; the human intervention is required only for tasks that cannot be automated by machines. Second, the process should be scalable to study other different architectures and programming languages, even potential ones in the future.

To this end, we have designed and implemented an evaluation system prototype, BINSIGHT, to systematize our evaluation process in an automated and scalable manner. We first describe the architecture of BINSIGHT (§ III-A), and then present each component including compilation (§ III-B), optimization (§ III-C), measurement (§ III-D), and root cause analysis (§ III-E).

A. Architecture

BINSIGHT is designed with the key principle of modularity and extensibility, so that it is straightforward to make modifications to accommodate different needs, such as adding new test projects, extending to new program languages and architectures.

Based on this design principle, we present, in Figure 2, the architecture of our evaluation system prototype, comprising four key modules: 1) compilation, 2) optimization, 3) measurement, and 4) root cause analysis. First, the compilation module (①) takes as input the test sets, compiles and links them with relocation information according to the user-specified configuration, yielding an unoptimized binary. Second, the optimization module (②) receives the unoptimized binary and samples its profile data by executing it. Then the profile data guides the optimizer to optimize the target binary and generate an optimized version. Third, the measurement module (③) measures multiple performance metrics for both the unoptimized and optimized binary. Finally, the root cause analysis module (④) conducts necessary case studies to uncover reasons for binary performance and optimization failure, propose best practices and suggestions, then produces empirical results.

B. Compilation

The compilation module compiles and links source code into a binary file containing relocation information according to the configuration user-specified, which serves as the baseline for subsequent modules. Although the compile option can be added or removed from the configuration based on actual needs, the *emit relocations option* (`--emit-relocs` or `-q` option in most linkers) should not be omitted as BOLT requires relocation information and enabling it to optimize and alter the position of functions [1].

Unfortunately, mainstream linkers like BFD [51], Gold [52] and lld [53], all support the aforementioned option, but not all do. Thus this module has to compromise on the compiler used. For example, `gc`, an official compiler for Go, does not support that option due to it using a self-build toolchain. There are other two implications of Go compiler: gccgo [54] and gollvm [55], but the development of gollvm has not been active in recent years. For this reason, we use gccgo in the following evaluation which the latest version has implemented go1.18 [56]. Other languages whose official compiler's backend is based on the GCC or LLVM toolchains, such as Rust, don't have the same issue because they are using BFD or lld linker.

C. Optimization

This module optimizes the target binary and produces the optimized version by leveraging profile data collected at runtime, following these steps: 1) Profiling: the target binary is run and using performance tools like `perf` [57] to sample the cpu-cycles event and taken branch stack if supported by corresponding architecture. In order to reduce the influence of the noise and obtain more accurate profile data, some essential measures should be taken such as killing irrelevant processes, increasing process priority and setting the CPU affinity, as well as repeating multiple times according to configuration. 2) Optimization: by leveraging the profile data, the binary is performed a series optimization pass by BOLT with the options declared in the configuration.

It should be pointed out that the taken branch stack is critical information for profile-guided optimization as well as BOLT. It records a series of consecutive taken branches, providing accurate counts for critical edges and improve the resilience to lousy sampling [1]. For this reason, it is essential to enable it if the corresponding platform supports it, otherwise the optimization will not achieve the maximum performance improvement.

D. Measurement

In this module, we focus on measuring performance metrics for both unoptimized and optimized binaries using `perf`

where the metrics of unoptimized version as the baseline for comparison. The concerned metrics can be specified in the configuration and include:

CPU cycles. This metric represents the number of clock cycles spent running the program, and is a straightforward reflection of the its speed. The primary purpose of binary optimization is to reduce this metric.

Branches and branch misses. This metric indicates the effect of branch prediction after basic block reordering, and the effective reordering should reduce branch misses.

Cache and cache misses. This metric consists of both I-cache and D-cache. BFDO aims to improve instruction locality, including reducing I-cache occupation, grouping hotspot code, and folding identical code. Some of them are the trade-off between I-cache and D-cache. This metric reflects the effect of I-cache optimization and its influence on D-cache.

TLB and TLB misses. Similar to cache, this metric also reflects instruction and data locality. But the scope it concerned is on the page level that is bigger than cache size, so it is useful for indicating the locality of large programs.

To reduce noise impact, the measurement should be repeated multiple times, and then use the equation (1) to calculate the average speed up results.

$$speedup = \frac{\sum^{repeat} \frac{unopt-opt}{unopt}}{repeat} \quad (1)$$

E. Root Cause Analysis

Based on measurement results, this module analyzes root causes of both binary performance and optimization failure, and then suggests avoidance and refinement strategies. Prior research inspired us that binary optimization might suffer potential problems such as disassembly [20] [21] [58], profiling [59] [60], relocation [61], etc. Therefore, we incorporated root cause analysis in BINSIGHT, to identify deeper reasons for both positive and negative cases. First, for optimized binary performance, we analyze degradation reason from the perspective of the particularities of programming languages and hardware limitations, and further explore most impactful factors for performance improvement. Second, for failed cases, we investigate failure reasons and point out possible deficiencies in the optimizer. Finally, based on our analysis, we provide best practices for BFDO usage and suggestions to refine its ecosystem and produce empirical results.

IV. EVALUATION

In this section, we present experiments to evaluate BFDO across multiple languages, multiple architectures and compiler optimization neutrality. We first present the research questions guiding the experiments (§ IV-A), the benchmark we used(§ IV-C) and the experimental results of BFDO (§ IV-D to § IV-F).

A. Research Questions

By presenting the experimental results, we primarily investigate the following research questions:

TABLE II: Experimental setup

	Hardware	OS	Compiler	Optimizer
x86-64	Intel i7-12700K 128GB RAM	Ubuntu 22.04	clang 18.1.1 rustc 1.77.1 gccgo 13.2.0 mlton 20231123	BOLT 18.1.1
AArch64	BCM2712 CPU (4x Cortex-A76) 8GB RAM	Ubuntu 22.04	clang 18.1.2 rustc 1.78.0 gccgo 13.2.0 mlton 20231123	BOLT 18.1.2
RISC-V	TH1520 SOC (4x Xuantie C910) 16GB RAM	Debian 12	clang 18.0.0 rustc 1.70.0 gccgo 13.2.0	BOLT 18.1.2

RQ1: Diverse Programming Languages. Is BFDO still effective on binaries compiled from different programming languages? What are the performance changes for these binaries? What are the shortcomings for these binaries?

RQ2: Multiple Architectures. Is BFDO still effective on different architectures? What are the performance change on those architectures? What are the shortcomings on those architectures?

RQ3: Compiler optimization neutrality. Are BFDO performances affected by compiler or linker optimization? What is the impact?

B. Experimental Setup

We conducted experiments on three current representative architectures: x86-64, AArch64 and RISC-V. The experimental environment of each machine is as shown in Table II. Since Intel CPU has 8 P-cores and 4 E-cores with different profiling functions, so all experiments were restricted to running on P-core. Additionally, as official OS for RISC-V machine only support Debian, so we used a different OS version, this is negligible as our experiment had few dependency on specific OS features. For mlton, the SML compiler we used, as its latest version does not support RISC-V, we only conducted our SML experiments on x86-64 and AArch64, and we use the LLVM backend for code generate.

For all experiments, we use BOLT to optimize their I-cache locality with the following options:

```
-reorder-blocks=ext-tsp -reorder-functions=hfsort
-split-functions -split-all-cold -split-eh -dyno-stats
```

We gathered profiles using `perf record` with options `-e cycles:u -j any,u`. We then used `perf2bolt` tool to transform the profiles to BOLT format, with a `-nl` option if the corresponding architecture did not support branch stack sampling.

C. Datasets

We used two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world workloads.

Micro-benchmarks. To simplify the question and quickly verify our hypothesis, we first construct a set of micro-benchmarks with the simple and fast build and test process.

All these test cases are small traditional algorithms, both being written in in C++, Go, Rust and Standard ML (SML), and compiled with default compiler optimization (`-O0`). These benchmarks is usually running very fast, so for ease of measurement we amplified their running times by looping them thousands of times.

Real-world workloads. To reflect the application of different programming languages in their respective fields, we selected diverse real-world open-source programs for each language from GitHub. Unfortunately, few real-world workloads are written in SML, so we could not further evaluate this language. To select the appropriate datasets, we had two key criteria: 1) language skilled and 2) performance sensitive. First, we used programming languages as keywords to search open-source projects, and sorting by star count in descending order. We believe projects with high star numbers usually reflect the scenarios the related languages are proficient in. Second, we only selected projects prioritizing executing performance, preferring those with end-to-end benchmarks. These projects have higher optimization value and are designed to be executed frequently. A slight performance degradation may be magnified infinitely, as well as improvements.

Although many excellent projects should have been chosen, we have to forgo some due to compiler limitations described in section III-B.

D. RQ1: Diverse Programming Languages

To answer **RQ1** by demonstrating the effectiveness and efficiency of BFDO across multiple languages, we first extended the research with more programming languages and conducted experiments by applying BFDO on the datasets described. To control variables, we only present results of x86-64 in this section, and others will present in the following section.

Micro-benchmark: To simplify, we first conducted experiments on micro-benchmarks. As shown in Table III, column x86-64 presents performance results of the micro-benchmarks, all tests are successfully applied by optimizer without errors. Only two tests exhibited significant performance degradation (over 1.0%), while 73.3% of tests achieved performance improvement without optimization errors. All four languages achieved improved performance on average, and among them, Rust has the best optimization effect, with an average of 15.26% and a maximum of 35.89%, followed by Go, then C/C++ , and SML is the worst.

Real-world workload: After the experiment of the micro-benchmarks, we then evaluated the real-world workloads we selected via the approach described in section IV-C. Table IV, column x86-64 presents the results of the real-world workloads. The performance improvement of all languages on x86-64 reached 4% on average without errors, while real-world workloads of Go achieved maximum average optimization effect with 4.74%, followed by Rust, and C/C++ is the worst. Compare to the result of micro-benchmarks, whole real-world workloads have achieved performance improvement.

Support code influence.: Go and Rust demonstrated superior performance relative to C/C++ due to they have more cold support code within binaries, which BFDO can significantly benefit from. Go and Rust incorporate more complex runtimes than C/C++, emitting considerable support code into compiled binaries, including stack overflow check, array bounds check, Drop trait code, and numerous support library codes, separating hot code and inflating binary size. Such codes are rarely executed or only executed once for each function, but they affect the locality of instruction. For example, Go conducts a bounds check preceding array access. If violated, panic function is called, passing necessary parameters, which will take four instructions but is rarely executed. Similarly, Rust Drop trait code commonly executes before function exit, increasing the jump distance between hot code. In our micro-benchmark, on average, the hot code of C/C++ programs accounts for 43% of the code section size, while Go and Rust only account for meager 16% and 4%, respectively, with section size of 23x and 122x larger than C/C++, which indicates Go and Rust have greater optimization potential.

Machine code style: The differences in machine code style can also impact the effectiveness of BFDO, especially for functional languages, where optimization performances can be at either extreme. BFDO typically disassembly binary at functions granularity, which is determined by the granularity of relocation information. However, this granularity is not suitable for functional languages. In our observation, after continuation-passing style transformation, in the machine code level, SML organizes all user code into a single function (aka. Chunk). The original SML functions are accessed in Chunk via direct or indirect jumps rather than calls. The final binary contains a small number of functions, but each function is substantial in size. This not only increases the difficulty of function disassembly but also raises the penalty for disassembly failure. Therefore, SML faces a high risk of zero optimization compared to other languages.

E. RQ2: Multiple Architectures

To answer **RQ2**, investigating the effectiveness, efficiency and shortcomings of BFDO on different architectures, we then extended the research with more architectures and conducted an experiment by applying optimizer on the benchmarks on three representative architectures.

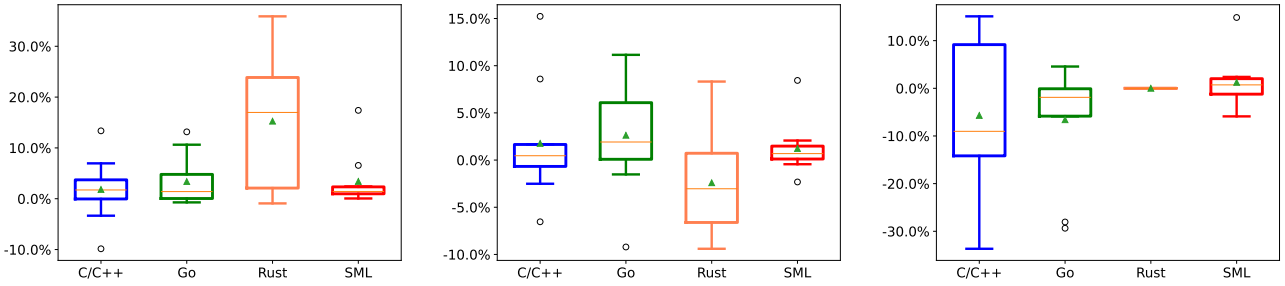
Table III presents the performance results on micro-benchmarks. Except Rust, there are achieved performance improvement in average on x86-64 and AArch64, and x86-64 has the largest improvement with an average of 15.7% higher than AArch64. Similar trends also occur in real-world workloads, as shown in Table IV, with over 4% average performance improvement on x86-64, and over 22% higher than AArch64.

BFDO encounters diverse errors on AArch64 and RISC-V, especially for Rust, only one case (RustPython) successfully optimized. Not only that, even when optimization succeeded, severe performance degradation is obtained on RISC-V on average. This indicates BFDO is not completely effective on AArch64 and RISC-V. To better answer **RQ2**, we then explore the root causes leading to optimization failures and

TABLE III: Performance for micro-benchmarks (% correct, per test case)

#	Case	x86-64				AArch64				RISC-V			
		C/C++	Go	Rust	SML	C/C++	Go	Rust	SML	C/C++	Go	Rust	SML
1	K-Means	0.45	-0.18	-0.43	0.96	-0.62	10.55	2.85	2.07	-11.3	-5.66	×	-1.47
2	Prim	0.94	-0.72	15.61	2.42	-2.51	1.54	-6.87	1.56	-8.39	-1.3	×	1.55
3	KMP	13.36	1.55	1.65	1.36	15.23	11.15	0.25	0.06	-33.67	4.57	×	-5.89
4	Matrix chain order	-9.85	-0.22	35.89	6.56	0.81	-9.21	-8.15	0.28	-18.24	3.03	×	-2.07
5	Max submatrix sum	3.73	1.88	20.2	0.99	1.67	-0.04	-5.84	-0.44	15.1	-5.90	×	1.27
6	Merge sort	3.68	0.71	25.08	0.72	-0.69	0.45	-4.36	1.09	-15.13	-1.01	×	2.18
7	N queen	2.51	5.76	-0.92	17.39	8.59	6.63	8.32	8.44	13.07	-29.36	×	14.86
8	Qucik sort	-0.21	1.26	18.35	0.05	1.6	-1.52	-1.72	1.24	13.84	0.21	×	0.17
9	sha256	-3.34	13.16	3.43	2.07	0.11	4.44	0.88	-2.32	-2.55	-28.05	×	2.39
10	TSP	6.96	10.63	33.73	1.5	-6.54	2.29	-9.40	0.27	-9.66	-2.49	×	-0.5
	Average	1.82	3.38	15.26	3.4	1.76	2.63	-2.4	1.23	-5.69	-6.60	—	1.25
	Variance	33.68	21.35	173.04	27.37	33.05	32.52	30.94	7.93	225.95	131.96	—	29.01

[1] The character × means optimization failure.



(a) x86-64 (b) AArch64 (c) RISC-V

Fig. 3: Box plots of micro-benchmark performance

TABLE IV: Performance for real-world workloads (% correct)

Language	Workload	x86-64	AArch64	RISC-V
C/C++	John	0.08	-0.46	3.87
	QuickJS	1.36	-0.28	⊗
	Redis	11.35	3.56	⊗
	Cppcheck	3.88	13.46	×
	Average	4.17	4.07	—
Go	gjson	5.58	9.28	-1.61
	goquery	1.24	1.08	-3.71
	websocket	1.69	1.46	18.41
	web framework	7.97	5.43	⊗
	grpc-go	7.24	-0.57	⊗
	Average	4.74	3.35	—
Rust	Rocket	5.09	2.74	×
	RustPython	2.47	2.41	⊘
	wasmtime	2.84	⊗	⊘
	hyper	2.01	⊗	×
	tantivy	9.42	0.3	×
	Average	4.36	—	—

[1] ⊘ compilation error, ⊗ disassembly error, × relocation error.

```
000000000073d738 <.L1^B1>:
  73d738:    01a0000b          .insn 4, 0x01a0000b
  73d73c:    0440006f          j     73d780 <.Llrsc_0>
  73d740:    00000013          nop

000000000073d780 <.Llrsc_0>:
  73d780:    0246000b          .insn 4, 0x0246000b
  73d784:    1606272f          lr.w.aqrl   a4,(a2)
  73d788:    01a0000b          .insn 4, 0x01a0000b
  73d78c:    00000013          nop
```

Fig. 4: RISC-V assembly code example that BOLT is unable to disassembly.

bad performance, and identified four key reasons: 1) customize instruction, 2) relocation exception, and 3) biased profile, 4) failed prediction.

Customize Instruction: Disassembly failures on RISC-V mainly caused by customize instructions in both micro-benchmarks and real-world workloads. RISC-V is an open-source ISA that allows chip implementers to add diverse extension instructions, and customize instruction provides a method to support those extension instruction with little modification to the compiler backend. On the other hand, customize instruction also allows using the nonexistent instruction simulated by software which is useful for implement specific instruction function and prototype design of instruction set. Although customize instruction brings flexibility, it dose pose a challenge to disassembly of BOLT.

As shown in Figure 4, disassembling from `grpc-go`, it is unable to disassembly the customize instructions declared by `.insn` directive where the value `0x01a0000b` and `0x0246000b` encode the instruction `SYNC.I` and `DCACHE.CVAL1` respectively on Xuantie C910 processor

```
1 foo:
2   auipc t0, %pcrel_hi(bar)      # R_RISCV_PCREL_HI20
3   addi  t0, t0, %pcrel_lo(foo) # R_RISCV_PCREL_LO12_I
4   nop
5 bar:
6   nop
7
8 label:
9   # expand by la.tls.gd a0, symbol
10  auipc a0,0       # R_RISCV_TLS_GD_HI20 (symbol)
11  addi  a0,a0,0    # R_RISCV_PCREL_LO12_I (label)
```

Fig. 5: Example of a RISC-V *PC-relative* relocation

[62]. In our experiment, the appearance of those customize instruction derives from the usage of synchronization primitives in programs. Xuantie C910 implemented the extension instruction sets of synchronization and cache, such as the above `SYNC.I` and `DCACHE.CVAL1`, which will be emitted by the assembler when assembling the atomic instruction compiled from the synchronization primitives.

Relocation Exception: The optimizations of Rust test sets were failed in all micro-benchmarks and most of real-world workloads on RISC-V, these are all due to the exception that the optimizer was unable to find the corresponding `%pcrel_hi`. This exception stems from the relocation type *PC-relative TLS GD GOT reference* in RISC-V. In some cases, the program would like to load the address of a label to a register (`la` or `lla` pseudo directive). The common way to do that in most architectures is using the PC-relative addressing. RISC-V also has this addressing mode but it requires two instructions, as shown as line 1 to 6 in Figure 5. The modifier `%pcrel_hi` indicates the high 20 bits of relative between PC and `symbol`, and the `%pcrel_lo` indicates the low 12 bits. In this example, to load the address of symbol `bar` into register `t0`, `auipc` instruction takes the high 20 bits of the relative address between PC (also the symbol `foo`) and symbol `bar` as source operand HI, storing $PC+(HI << 12)$ into register `t0`. Then `addi` instruction takes register `t0` and the low 12 bits of the relative address between symbol `foo` and symbol `bar` as source operands, adding them and storing the result to register `t0`.

Loading the address of a label also occurs in the global dynamic thread local storage (TLS) variable. To address this, RISC-V provides global dynamic TLS model that is used for PIC shared libraries and handles the case where more than one library uses thread local variables, and additionally allows libraries to be loaded and unloaded at runtime using `dlopen`. In this model, the address of a variable can be loaded using the `la.tls.gd` pseudoinstruction, which will expands to the assembly instructions and relocations as shown as line 8 to 11 in Figure 5.

In our experiment, BOLT do not provide the detailed error reasons of relocation exception. But by analyzing the number of the relocation type and error message, we found that this exception was caused because BOLT is unable to resolve `R_RISCV_TLS_GD_HI20` relocation type. The relocation of

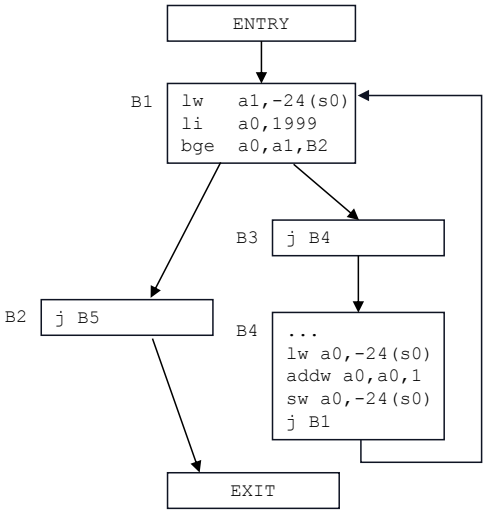


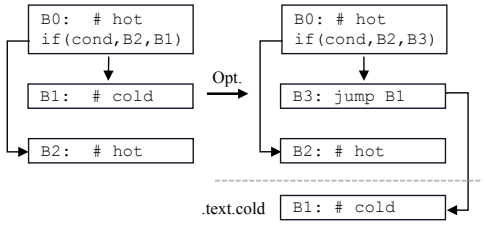
Fig. 6: Example control-flow graph for biased profile from a for loop generate by clang++ on RISC-V

this type is very common in Rust programs even for non-multithread programs while rare in C/C++ and Go, and we think this is caused by the complex runtime of Rust.

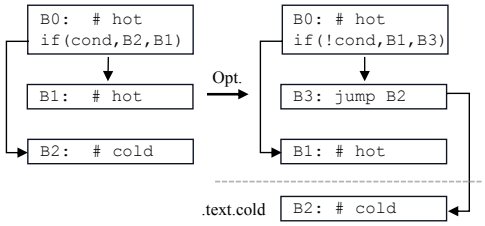
Biased Profile: The micro-benchmarks achieved significant performance degradation on RISC-V, we found it is due to the biased profiles. Sample-based profiling is do not a reflection of the real executing frequency of the program's basic block, it is affected by many factors, such as sampling frequency, basic block size and sampling environment. Among them, the basic block size is the most difficult factor to avoid by adjusting sampling options, and it is also the factor with the greatest impact. Sample-based profiling is usually done in units of instructions (or clocks), which means that large basic blocks containing more instructions have a greater probability of being sampled, while small basic blocks are rarely sampled. Although the probability of small basic blocks being sampled can be increased by increasing the sampling frequency, this will also increase the sampling probability of large basic blocks, resulting in the final ratio of the two frequencies tending to the ratio of the number of instructions.

Figure 6 is a for loop example compiled from the micro-benchmark using clang++ with `O0` optimization level, which containing 5 basic blocks and `B1`, `B3` and `B4` with the same executing count. In our experiment, since `B3` only contains one instruction, its sampling times are too few, and it is eventually mistaken for a cold basic block, while `B1` and `B4` are considered the hot basic blocks. After optimization, the hot/cold split algorithm split them into the different section that far apart in address space. This causes each loop to have to jump twice between the hot and cold section, which causes the severe degradation of i-cache locality, and in the worst case leads a simple for loop to a 1.5x increase in running time.

Failed Prediction: In addition to biased profile, another reason of the performance degradation on RISC-V stems from the failed branch prediction. After reordering basic block, the



(a) Case 1



(b) Case 2

Fig. 7: Example of failed branch prediction of condition statement on RISC-V

TABLE V: Max jump range on three architecture

jump type	x86-64	AArch64	RISC-V
unconditional	±2GB	±128MB	±1MB
conditional	±2GB	±1MB	±4KB

jump target of branch instruction will be corrected, but the inappropriate corrections may not improve performance, or may even cause performance degradation. In order to improve the code locality and the accuracy of branch prediction, hot basic blocks are reordered to be reached in a fallthrough manner, and the cold basic blocks are split into the cold section. However, the reordered basic block may be too far away and beyond the addressing range (±4KB) of the conditional jump instruction that reach it. To address this, the optimizer will insert an unconditional jump instruction after the conditional jump, but this inappropriate approach incurs a significant branch prediction failure.

Figure 7 illustrates the aforementioned situation. In case 1, the branch prediction will take cold block `B1` as the predicted branch, which is not an efficient layout. After optimization, the cold block `B1` is split into the cold section, and the original position of `B1` is inserted as an unconditional jump instruction to `B1`. Although it improves the code locality in case 1, it does not improve the branch prediction accuracy. In case 2, the situation is even worse. To be able to jump to cold block `B2`, the optimizer reverses the condition of the conditional jump instruction and insert an unconditional jump before `B1`, which will incur a branch prediction failure for each jump to `B1`.

When the improvement in code locality cannot make up for the overhead of branch prediction failure, the overall performance will degrade. Through our experiments, the failure of branch prediction in an if-else statement will incur a degradation of about 6%. These problems did not occur on x86-64 and AArch64. The addressing range of conditional jump instruction on x86-64 is up to 4GB, so it can reach the cold block directly in a single instruction without affecting branch prediction. Although the addressing range of AArch64 conditional jump instruction is not so large range like x86-64, it inserts the unconditional jump after the hot block instead of before. Since we cannot decide the approach of the jump target correction of BOLT, so we cannot evaluate the specific performance if using effective measures. But we believe this is why binary optimization performs so poorly on RISC-V.

F. Compiler optimization neutrality

In this section, to answer **RQ3** by investigating compiler optimization neutrality, we conducted an experiment by applying binary optimization on the micro-benchmarks with two compiler optimization options inspired from the previous experiment, including 1) compiler optimization level and 2) linker relaxation.

Compiler optimization level: Binary optimization performance is affected by the compiler optimization level. Instead of achieving greater performance gains, binaries with low optimization levels may cause performance degradation. In reviewing Figure 6, one of the factors contributing to the loop inefficiencies is the biased profile, while the other is the excessively low optimization level, which produces a complex CFG with the tiny basic block. To avoid this, the best way is to increase the compiler optimization level, using the compiler pipeline to simplify the CFG and reduce the unnecessary small basic block, such as the `j` instruction in the example, before binary optimization. Our experiment found that measure will reduce the average performance of the C/C++ micro-benchmarks on RISC-V from -5.69% to -1.94%.

Linker Relaxation: Cppcheck encounters the relocation exception that `R_RISCV_JAL` relocation is out of its address range, which is caused by the linker relaxation optimization. The linker relaxation optimization RISC-V introduced is a technique to reduce code size. In RISC-V, as well as some RISC, the max instruction length is only up to 32 bits, so it impossible to address the whole memory range using a single instruction, especially on 64-bit machines. To this end, RISC machine usually use two instructions to complete addressing if the symbol is too far, for example, the first instruction locate the page range, and the second determine the offset within page. The symbol location is not yet determined in compilation stage until in linker stage, so compiler will generate two instruction to ensure sufficient addressing space, and generate a relaxation relocation item to suggest linker to replace two instructions with a single one if the symbol is not far actually and can be addressed by a single instruction.

This optimization is based on the assumption that the symbol location will not be changed after link, but the binary optimization breaks that assumption. The function reordering, basic block reordering, and the hot/cold code split all will change the symbol location, so it is possible that the symbol distance out of the range of the relaxed instruction and causes the relocation process to fail, especially for the binary with

large code size. In Cppcheck, its code section size exceeds 6MB, but the relaxed instruction `jal` has only 2MB addressing capacity, then the relocation error was caused when any symbol distance from `jal` out of 2MB. To avoid this, we recompile Cppcheck with disabling the relaxation optimization (using `--no-relax` option), then it is optimized without error and achieves speed up of 21.02%.

V. Best Practice and Further Direction

In this section, we present four best practices for binary optimization to use it and avoid bad performance, and figure out the improvement measures for both binary optimizers and compilers.

A. Support relocation information

In order to use binary optimization, the first need is to pick a suitable compiler with the ability to export relocation information. Binary optimization needs precise disassembly and relocation information, which requests the compiler have the ability to export symbol table, and relocation information to the binary file. Any compiler supports the symbol table, but not all support relocation information, such as `gc`. This may require some changes to the source code, as not all compilers support the same feature, such as gccgo not yet supporting generic that is available in `gc`. Besides, optimizer support for relocation type is still limited, such as `R_RISCV_TLS_GD_HI20`, so using unsupported relocation information should be avoided. Those is a trade-off between performance and program implementation.

Ideal optimization should be transparency to the program, the aforementioned measures are too expensive to modify the exciting program or avoid the usage of unsupported features. This inspire us that it is essential to enhance the compiler to support exporting relocation information, extending the optimizer to support more relocation types. Furthermore, recent compilers commonly hide relocations within a single compilation unit. So a aggressive idea to support binary optimzition is to be able to emit those relocations.

B. Don Not Use On O0 binary

Binary optimization is a supplement to traditional compilation optimization and cannot completely replace them. Blind replacement cannot achieve greater performance and may also cause performance degradation. Unoptimized binary by compiler (`O0`) usually has many tiny basic block and complex jump structure, which is more susceptible to biased profile, especially on RISC-V. On the other hand, compiler optimization pipelines are the cheaper and proven way to gain performance than binary optimization does. So the appropriate way is using compiler optimization first to simplify program structure and apply other optimizations, and considering binary optimizations as a mean to further improve performance. By applying the compiler optimization first on our micro-benchmarks, the performance degradation was reduced by an average of 4.57% on RISC-V.

C. Use Linker Relaxation With Caution

Although linker relaxation can reduce the binary size and improve the performance on RISC-V, the improvement is negligible compared to binary optimization, but it will have a fatal impact on binary optimization, especially for large binary. We suggest that the better way is to turn off the linker relaxation when compilation. If the linker relaxation is really necessary for you, you can keep linker relaxation but disable it when you encounter the relocation error of out of the addressing range. By disabling linker relaxation, we avoid the failure of Cppcheck.

The trade-off between linker relaxation and binary optimization is a situation that should not exist, because it can be addressed by rewriting the instruction by the binary optimizer. To be specific, if the addressing range is out of the bound of the single instruction, the binary optimizer should replace the single instruction with two instructions with larger addressing range, which is the reverse process of linker relaxation. We found this was integrated by BOLT, but regrettably it doesn't work completely correctly.

D. Ensure Sampling Accuracy

Sample-based profiling is not a real reflection of the executing count of basic block, the sampling count is related to the instruction number of basic block. If the optimization does not achieve the desired performance, make sure the enough compiler optimization is applied first, and try to adjust the sampling frequency to improve the accuracy, such as using `-F` option for `perf`. If necessary, use instrumentation-based profiling instead of sample-based, which is an accurate profiling method, but will causes obvious performance costs.

For binary optimizer, it is responsible to take some measures to improve its resilience to the bad profile data, rather than blindly trusting it, such as combining control flow static analysis to infer the execution number of the basic block, deciding the reorder weight based on both profile data and basic block size, and giving the warning messages to remind the rise of performance degradation when necessary. For chip designer, providing powerful profiling hardware is also beneficial.

VI. Discussion

The architecture-specific characteristics of binary feedback-directed optimization are destined to be challenged by the architecture. This is because it has to complete two crucial steps. First, in order to rewrite the binary, it must disassemble the binary correctly to construct the control-flow graph. Disassembly is an NP-complete problem and has to use some heuristics, such as relying on the relocation information. Second, it should obtain an accurate profile with as minimal performance costs as possible. Some of the architectures, such as AArch64 and RISC-V, have limited support for hardware profiling. What is worse is that the virtual cloud hosts that are currently widely used do not even support basic hardware sampling.

Different programming languages and compilers also have the effect on binary optimization. Diverse programming language characteristics and code generation methods will affect the structure of the final binary, which will cause different performance changes and relocation problems.

This paper investigates binary feedback-directed optimization in a comprehensive perspective, which demonstrates that binary optimization is not yet completely effective and efficient. We propose some best practices for using binary optimizer and the suggestions for optimizer and compiler, and we hope those will be helpful to refine the binary optimization ecosystem. But we still have some limitations in our work, which need further research.

First, we found the binary size increased significantly after optimization. This is because the code section aligns to huge page (2MB) and BOLT will duplicate the old code section to handle functions that cannot be disassembled, which results in the average 75x and 0.8x increase in micro-benchmark and real-world overload size. Huge page alignment can be disabled through the option, but the duplicated code section is not. We believe the best way is to duplicate the code if necessary, such as when the function that cannot be disassembled really exists. Second, our work does not involve instrumentation-based profiling, it is necessary if the runtime environment does not support the sample-based profiling, such as on the virtual cloud machine.

VII. Related Work

There are a significant amount of studies on binary optimization and profiling.

Bianry optimization. There are many excellent studies have been conducted on binary optimization. Panchenko et al.[1] presented a static binary optimizer BOLT for data-center applications, which demonstrate that profile data can be used more precisely on binary level. Williams-King and Yang et al. [5] implemented CodeMason that performs static binary rewriting based on a binary rewriting platform called Egalito[63]. Panchenko et al.[2] demonstrated Lightning BOLT that based on the work of BOLT. They addressed the CPU and memory overhead of BOLT by introducing parallel processing and selective optimizations. Zhou and Jones et al. [64] presented a framework called Janus to address the challenge of automatic binary parallelisation. Savage and Jones et al. [35] proposed a post-link profile-guided optimization tool called HALO to improve the layout of heap data to reduce cache misses automatically. However, their work mainly focuses on the x86 CPU architecture and the binary compiled from C/C++. Our work studied the effectiveness and performance improvement of other platforms and programming languages, aiming to fill in this gap.

Profiling technique. Profiling technique is a key component of binary optimization. Instrumentation-based profiling is a common way to collect precise profile data, but may cause nonnegligible runtime overhead. To mitigate the cost, Ball et al.[65] optimized the probe placement using minimal spanning tree. Additional, Cho et al.[66] proposed a novel instrumentation framework which reported an average 3% to 6% runtime slow down. Since the overhead of instrumentation-based profiling is still underirable for production deployment, sample-based profiling is widely adopted by many tools. He et al. [10] proposed pseudo-instrumentation technique as the supplementary of sample-based profiling to improve profile quality without incurring the overhead of traditional instrumentation. Moreira et al. [50] improved static profiling technique called Evidence-Based Static Prediction (ESP) proposed by Calder et al. [49], which enables the output profiles to be used for binary optimization.

VIII. Conclusion

In this work, we presented the first and most comprehensive study of the binary optimization in the context of BOLT. By designing and implementing a software prototype, we investigate the effectiveness and efficiency of binary optimization on three representative architectures and programming languages. We proposed root causes leading to optimizer failures. We also revealed reasons for hurting performance. We provided suggestions to compiler developers, optimizer developers, and chip designers. A consideration of them can refine the future binary optimization ecosystem.

References

[1] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "Bolt: A practical binary optimizer for data centers and beyond," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Washington, DC, USA: IEEE, Feb. 2019, pp. 2–14.

[2] M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning bolt: Powerful, fast, and scalable binary optimization," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. Virtual Republic of Korea: ACM, Mar. 2021, pp. 119–130.

[3] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, "Propeller: A profile guided, relinking optimizer for warehouse-scale applications," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vancouver BC Canada: ACM, Jan. 2023, pp. 617–631.

[4] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney, "Ispike: A post-link optimizer for the intel/spl reg/ itanium/spl reg/ architecture," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 15–26.

[5] D. Williams-King and J. Yang, "Codemason: Binary-level profile-guided optimization," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST'19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 47–53.

[6] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming hardware event samples for precise and versatile feedback directed optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 376–389, Feb. 2013.

[7] D. Chen, D. X. Li, and T. Moseley, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. Barcelona Spain: ACM, Feb. 2016, pp. 12–23.

[8] D. Novillo, "Samplepgo - the power of profile guided optimizations without the usability burden," in *2014 LLVM Compiler Infrastructure in HPC*. LA, USA: IEEE, Nov. 2014, pp. 22–28.

[9] "[samplefdo] flow sensitive sample fdo (fsafdo) profile loader," https://reviews.llvm.org/D107878.

[10] W. He, H. Yu, L. Wang, and T. Oh, "Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Mar. 2024, pp. 322–333.

[11] R. Wang, D. Gibson, K. Rodrigues, Y. Luo, Y. Zhang, K. Wang, Y. Fu, T. Chen, and D. Yuan, "µslope: High compression and fast search on semi-structured logs."

[12] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, Í. Goiri, S. Elnikety, R. Fonseca, and A. Belay, "Making kernel bypass practical for the cloud with junction."

[13] L. Yu, X. Zhang, H. Zhang, J. Sonchack, D. Ports, and V. Liu, "Beaver: Practical partial snapshots for distributed cloud services."

[14] "Tiobe index," https://www.tiobe.com/tiobe-index/.

[15] "New case studies about google's use of go."

[16] "Language details of the firefox repo," https://4e6.github.io/firefox-lang-stats/.

[17] "Rust for linux," https://rust-for-linux.com.

[18] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Securing the device drivers of your embedded systems," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, Aug. 2019.

[19] "Documentation - the go programming language," https://go.dev/doc/.

[20] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 583–600.

[21] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 401–414.

[22] "Embedded trace macrocell architecture specification," https://developer.arm.com/documentation/ihi0014/q/Introduction/About-Embedded-Trace-Macrocells.

[23] A. Waterman, K. Asanovic, J. Hauser, and S. Inc., "The risc-v instruction set manual volume ii: Privileged architecture document version 20211203," *CS Division,EECS Department, University of California, Berkeley*, Dec. 2021.

[24] S. Chen and H. Wang, "Compiler support for linker relaxation in risc-v."

[25] C. T. Apple, "Evaluation and performance of computers: The program monitor—a device for program performance measurement," in *Proceedings of the 1965 20th National Conference On -*. Cleveland, Ohio, United States: ACM Press, 1965, pp. 66–75.

[26] D. E. Knuth, "An empirical study of fortran programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, Apr. 1971.

[27] D. E. Knuth and F. R. Stevenson, "Optimal measurement points for program frequency counts," *BIT*, vol. 13, no. 3, pp. 313–322, Sep. 1973.

[28] "Gcc support for pgo," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[29] "Llvm support for pgo," https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization.

[30] A. Asthana, "Speed up windows php performance using profile guided optimization (pgo)," https://devblogs.microsoft.com/cppblog/speed-up-windows-php-performance-using-profile-guided-optimization-pgo/, May 2013.

[31] R. Lander, "Conversation about pgo," https://devblogs.microsoft.com/dotnet/conversation-about-pgo/, May 2021.

[32] P. Yuan, Y. Guo, and X. Chen, "Experiences in profile-guided operating system kernel optimization," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. Beijing China: ACM, Jun. 2014, pp. 1–6.

[33] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An optimizer for alpha/nt executables."

[34] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of win32/intel executables using etch."

[35] J. Savage and T. M. Jones, "Halo: Post-link heap-layout optimisation," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 94–106.

[36] "Llvm support for bolt," https://github.com/llvm/llvm-project/tree/main/bolt.

[37] "Optimizing clang with llvm bolt," https://github.com/llvm/llvm-project/blob/main/bolt/docs/OptimizingClang.md.

[38] M. Panchenko, "Optimizing the linux kernel with llvm bolt."

[39] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 196–205, Jun. 1994.

[40] N. Nethercote and J. Seward, "Valgrind," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44–66, Oct. 2003.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[42] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 357–390, Nov. 1997.

[43] N. Gloy, Z. Wang, C. Zhang, B. Chen, and M. Smith, "Profile-based optimization with statistical profiles."

[44] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, no. 2, pp. 187–206, Apr. 1996.

[45] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM SIGPLAN Notices*, vol. 32, no. 5, pp. 85–96, May 1997.

[46] I. Corporation, "Intel® 64 and ia-32 architectures software developer's manual," no. 325384-083US, Mar. 2024.

[47] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "Ts-perf: General performance measurement of trusted execution environment and rich execution environment on intel sgx, arm trustzone, and risc-v keystone," *IEEE Access*, vol. 9, pp. 133 520–133 530, 2021.

[48] J. M. Domingos, P. Tomas, and L. Sousa, "Supporting risc-v performance counters through performance analysis tools for linux (perf)," Dec. 2021.

[49] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, pp. 188–222, Jan. 1997.

[50] A. A. Moreira, G. Ottoni, and F. M. Quintão Pereira, "Vespa: Static profiling for binary optimization," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 144:1–144:28, Oct. 2021.

[51] S. Chamberlain and I. L. Taylor, "The gnu linker," *Linker. pdf*, 1991.

[52] I. L. Taylor, "A new elf linker," in *GCC Developers' Summit*, vol. 2008. Citeseer, 2008, pp. 129–136.

[53] R. Ueyama, "Lld: A fast, simple and portable linker," in *LLVM Developer's Meeting*, 2017.

[54] "Top (the gnu go compiler)," https://gcc.gnu.org/onlinedocs/gccgo/.

[55] "Gollvm - git at google," https://go.googlesource.com/gollvm/.

[56] "Setting up and using gccgo - the go programming language," https://go.dev/doc/install/gccgo.

[57] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[58] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 627–642.

[59] "Tech-privileged@lists.riscv.org | a proposal to enhance risc-v hpm (hardware performance monitor)," https://lists.riscv.org/g/tech-privileged/topic/a_proposal_to_enhance_risc_v/75675990.

[60] L. Yan, D. Thompson, and L. Support, "Using perf on arm platforms."

[61] "Should we constrain the target label of a %pcrel_lo to be in the same section? · issue #90 · riscv-non-isa/riscv-elf-psabi-doc," https://github.com/riscv-non-isa/riscv-elf-psabi-doc/issues/90.

[62] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi, "Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension : Industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 52–64.

[63] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 133–147.

[64] R. Zhou and T. M. Jones, "Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2019, pp. 15–25.

[65] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16,

no. 4, pp. 1319–1360, Jul. 1994.

[66] Hyoun Kyu Cho, T. Moseley, R. Hank, D. Bruening, and S. Mahlke, "Instant profiling: Instrumentation sampling for profiling datacenter applications," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Shenzhen: IEEE, Feb. 2013, pp. 1–10.