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

Mingliang Liu, Shanlin Deng, and Baojian Hua(✉)

School of Software Engineering, Suzhou Institute for Advanced Research
University of Science and Technology of China, Suzhou 215123, China
{liumingliang, dengshanlin}@mail.ustc.edu.cn, bjhua@ustc.edu.cn

Abstract. Binary feedback-directed optimization (BFDO) is a novel technique for optimizing program binaries by leveraging dynamic profiles, showing promising potential in various domains including data centers, clouds, and embedded systems. However, state-of-the-art BFDO approaches primarily focus on specific instruction set architectures and programming languages, while overlooking the architectural discrepancies, language diversity, and compiler optimization options. Consequently, a comprehensive study of BFDO is still lacking, hindering its broader applications.

In this paper, to address this gap, we present the first systematic study of BFDO, providing insights into its current capabilities, key grand challenges, and future research opportunities. Specifically, we first conduct an empirical study using a novel end-to-end evaluation tool to assess the effectiveness of BFDO. Our analysis identifies three root causes of BFDO failures and highlights three factors influencing its optimization effectiveness. Based on these findings, we propose four best practices and outline three promising research opportunities, offering guidance for future advancements. Applying our best practices to RISC-V benchmarks improves performance by an average of 4.57%, effectively mitigating existing relocation failures.

Keywords: Compiler optimizations · binary feedback-directed optimizations · profile-guided optimizations

1 Introduction

Binary feedback-directed optimization (BFDO) [36] [37] [40] [32] [48] is a promising post-link compiler optimization that directly enhances a target program's binary by leveraging the dynamically collected runtime profiles. Since BFDOs can more accurately map runtime profiles to binaries [10] than to high-level program source code or compiler intermediate representations, they offer performance improvements beyond those typically achieved by static compilers or traditional profile-guided optimizations (PGOs) [9] [35] [31] [22]. For instance, BOLT [36], a recently proposed BFDO, achieved speedups of up to 7.0% for

data-center applications on x86-64. Over the next decade, the growing demand for greater program efficiency in data centers, cloud computing, and embedded systems, along with the need for lower energy consumption, will make BFDO an increasingly valuable optimization technique.

While existing BFDOs have demonstrated promising potential for achieving significant performance improvements in practical workloads, they remain limited in terms of architectural independence, language diversity, and compiler optimization neutrality. First, existing BFDO studies [36] [37] [40] [48] primarily focus on a single architecture, x86-64, while neglecting others. We speculate that this focus on x86-64 stems not only from its widespread adoption in data centers and cloud environments but also from its hardware functionalities, such as Last Branch Record (LBR), which are essential for generating precise runtime profiles. However, as RISC architectures like AArch64 and RISC-V are increasingly deployed, we argue that the advantages of BFDO on x86-64 do not necessarily extend to RISC architectures for several technical reasons. First, the static disassembly required for BFDOs remains an unsolved challenge [3], which is particularly difficult for RISC architectures due to the frequent intermixing of data and instructions [25]. Second, profiling hardware on RISCs exhibits significantly different capabilities [46], meaning that profiles generated by this hardware may lack the precision required for effective BFDO optimization. Therefore, a comprehensive understanding of BFDOs on RISC architectures is urgently needed.

Second, existing BFDOs primarily focus on specific languages such as C/C++, neglecting support for other emerging languages. Although C/C++ are widely used in cloud computing, data centers, and embedded systems [44] [19] [50], languages like Go and Rust are also gaining traction [43]. For instance, Go has been extensively adopted by vendors like Google and has facilitated the development of major open-source projects such as Docker and Kubernetes [38]. Similarly, Rust is being deployed in Firefox [34] and the Linux kernel [28]. Furthermore, binaries compiled from these languages exhibit distinct characteristics compared to those compiled from C/C++, due to differences in compilation techniques and runtime organizations. For example, Go's binaries contain extensive runtime information, including garbage collection mechanisms, while binaries from functional languages such as ML consist of numerous smaller basic blocks due to the use of continuations [4] [26] — both of which are absent in C/C++ binaries. Consequently, evaluating the effectiveness of BFDO across diverse programming languages and binary structures is crucial.

Third, existing BFDO studies [36] [37] [40] [32] [48] have largely overlooked the impact of compiler optimizations on target binaries, making it challenging for developers to tune optimization options for optimal BFDO performance. Since compiler optimizations are applied in the early compilation stage without explicitly considering the post-link binaries, they may inadvertently reduce the effectiveness of state-of-the-art BFDOs. For example, the default optimization level (`O0`) may hinder sample-based profile generation by producing a large number of smaller basic blocks, while linker relaxation optimizations [11] may interfere with function reordering by replacing call instructions with those of

narrower ranges (e.g., `jalr` to `jal`). Therefore, a comprehensive understanding of how compiler optimizations affect BFDO effectiveness is essential.

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, key grand challenges, and future research opportunities, providing insights into their continued development. To achieve this, we design BinInsight, a novel automated and scalable tool, to aid in our investigation, and employ both quantitative and qualitative methodologies on micro-benchmarks and real-world workloads. We then conduct a qualitative study to investigate BFDO failures and their root causes, current limitations and mitigation strategies, best practices, and future research directions.

Specifically, our investigation addresses the three aforementioned limitations of existing BFDOs. First, we study BFDOs on three representative architectures: x86-64, AArch64, and RISC-V. These architectures are chosen not only because they represent distinct application scenarios — including cloud, embedded, and mobile systems — but also due to their differing dynamic profiling capabilities, which are crucial for BFDO effectiveness. Second, we study binaries compiled from four programming languages: C/C++, Go, ML, and Rust. These languages are selected because they are not only widely adopted but also represent diverse programming paradigms, including imperative, object-oriented, and functional programming. Third, we experiment with various compile-time optimization options as well as link-time optimizations, such as linker relaxation. Moreover, thanks to the scalable design of BinInsight, our system is not restricted to those specific combinations and can be extended to explore potential combinations of architectures, languages, and optimization strategies.

Our study yields significant findings and insights. First, we identify three primary root causes of BFDO failures, originating from issues in disassembly and symbol relocation. Second, we reveal three key factors — compiler support code, biased profiles, and branch prediction failures — that substantially affect BFDO effectiveness. Based on our findings, we propose four best practices for using BFDOs, and point out three potential directions for future studies. By applying our best practices to RISC-V benchmarks, we achieve an average performance improvement of 4.57% over state-of-the-art while simultaneously preventing relocation failures.

To the best of our knowledge, this work represents the *first* step towards gaining a comprehensive understanding of state-of-the-art BFDOs. In summary, our work makes the following contributions:

- **Comprehensive study.** We systematically analyze state-of-the-art BFDOs using both quantitative and qualitative methodologies.
- **Findings and insights.** We identify three root causes for BFDO failures, and three factors affecting their effectiveness.
- **Best practices and suggestions.** We propose four best practices for improving BFDOs and outline three promising research directions.

– **Open source.** We open source our software prototype, benchmarks, test scripts, and evaluation results to promote open research at: https://doi.org/10.5281/zenodo.15058804.

2 Background

To be self-contained, in this section, we present the background knowledge for this work on Feedback-directed optimization (FDO) (§ 2.1), and profile generation (§ 2.2).

2.1 Feedback-driven Optimization (FDO)

History. Feedback-driven optimization (FDO), also known as Profile-guided optimization (PGO), has been extensively studied with a long history [27] due to its potential to leverage runtime profiles for guiding compiler optimization. Recently, with advancements in hardware support and more accurate profile utilization techniques, FDO has garnered renewed research interest with notable variants and applications (*e.g.*, AutoFDO [9], SamplingPGO [35], and CSSPGO [22]).

Workflow. FDOs employ a two-stage build process as shown in Fig. 1. In the first stage, the source code is compiled (①) to binary, which is then executed to obtain profiles (②). Specifically, two primary approaches are utilized for profile acquisition: instrumentation-based and sample-based. The instrumentation-based approach instruments the target program to collect important runtime information such as execution frequency of basic blocks, functions, and loops, whereas the sample-based approach leverages specialized hardware features such as x86-64 LBR to periodically sample instructions to approximate execution frequencies. Guided by profiles collected in stage one (③), the second stage compiles the source to binary again (④), to further improve program performance (*e.g.*, more precise inline decisions for frequently called functions).

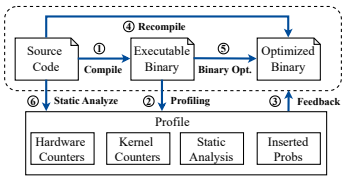


Fig. 1. Workflow of FDO and BFDO.

Traditional FDOs face two technical challenges. First, binary-only programs lack source code, making it difficult if not impossible to instrument sources. Second, compiler optimizations in phase (①) may dramatically change code layout, defeating the feasibility of *binary* profiles to *source* programs. Binary Feedback-directed Optimizations (BFDOs) [12] [32] [36] are introduced to address these challenges. As Fig. 1 presents, BFDOs directly instrument the executable binary to generate profiles which are then utilized to optimize the binary (⑤), without requiring program source code. Compared to FDOs, BFDOs (*e.g.*, HALO [39], Lighting BOLT [37], and Propeller [40]) often have a global view of the binaries

as they process post-link programs, enabling more aggressive optimizations including code layout, data compaction, hot/cold code splitting, alignment, and peephole optimizations.

Application. As a promising compiler optimization, FDO has been widely deployed in many applications. FDO is now supported by mainstream compilers [29], and has demonstrated effectiveness in optimizing diverse real-world programs (*e.g.*, Chrome [9] and Linux kernel [51]). Furthermore, FDO is also rapidly adopted by industry to optimize production applications in production environments. For example, FDO is used by Google [40] to optimize warehouse-scale applications, and by Meta to optimize compute-bound services [22].

2.2 Profile Generation

Profile generation is essential to perform effective FDO, as profiles directly guide optimization. Existing profile generation approaches can be classified into four categories: instrumentation-based, sample-based, hardware-based, and deep-learning-based.

Instrumentation-based. Instrumentation-based profile generation inserts instructions into the code to track accurate execution counts for various program paths, and has been deployed in many tools (*e.g.*, Atom [41], Clang [30], and BOLT [36]). While the instrumentation-based approach generates precise data, it introduces substantial runtime overhead as well as considerable code size increases to instrumented binaries, hindering its use in production scenarios.

Sample-based. Sample-based profile generation employs operating system mechanisms like clock interrupts to periodically capture program state snapshots as profiles. Traditional sample-based profiling is a statistical approximation which may produce inaccurate profiles. To mitigate this problem, precise sampling such as continuous profiling [2] [20] is proposed to significantly enhance the precision.

Hardware-based. Hardware-based profiling [1] [10] [14] collects profiles leveraging hardware performance counters, which critically rely on the underlying architectures. Fortunately, most modern architectures support performance counters (*e.g.*, Intel x86-64's Last Branch Records (LBR) [15], AArch64's Embedded Trace Macrocell (ETM) [5], and RISC-V's Hardware Performance Monitor (HPM), among others), which enable the generation of profiles with negligible runtime costs [42] [17].

Deep-learning-based. Deep-learning-based approaches generate profiles by estimating program branches statically. Once deemed impractical in the 1990s [7], deep-learning-based approaches have shown promising potential in optimizing binaries [33].

3 Approach

This section presents our approach to conducting the evaluation. Performing such an evaluation of various datasets on multiple platforms is challenging, due to two key reasons: automation and scalability. First, the evaluation must be

fully automated to efficiently evaluate a large number of projects; human intervention is only required for tasks that cannot be automated by machines, such as root cause analysis. Second, the evaluation must be scalable to study different combinations of architectures and programming languages, even potential ones not covered in this study.

To this end, we design and implement an automated software prototype, BinInsight, to aid in our evaluation. We first present an overview of BinInsight (§ 3.1), and then each component, including compilation (§ 3.2), optimization (§ 3.3), metric measurement (§ 3.4), and root cause analysis (§ 3.5).

3.1 Overview

BinInsight is designed with the key principle of modularity and extensibility, making it straightforward to accommodate different needs, such as adding new test projects, new programming languages, and architectures.

Fig. 2 provides an overview of BinInsight, which comprises four key components: compilation, optimization, measurement, and root cause analysis. First, the compilation component (①) takes the test cases as input, compiles and links them with relocation information according to the user-specified configuration, generating an unoptimized binary. Second, the optimization component (②) receives the unoptimized binary and samples the binary during execution to produce profiles, which guide the optimizer to optimize the target binary and generate an optimized version. Third, the measurement component (③) measures and records metrics for both the unoptimized and optimized binaries. Finally, the root cause analysis module (④) enables root cause studies for failures, upon which best practices and suggestions are composed.

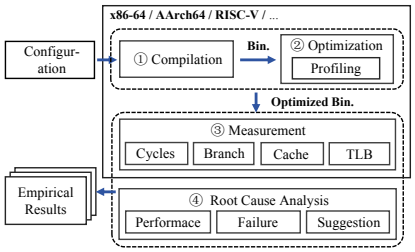


Fig. 2. An overview of BinInsight's workflow.

3.2 Compilation

The compilation module compiles and links source code into binaries with relocation information according to a user-specified configuration. A critical aspect of this process is selecting toolchains that support the link options required by BFDOs [36]. Specifically, the compiler toolchain must support *emit relocations option* (*i.e.*, `--emit-relocs` or `-q` in most linkers). However, we have observed that while mainstream linkers (*e.g.*, BFD, Gold, and lld) generally support the relocations option, some do not. For instance, `gc`, the official Go compiler, lacks support for this option whereas other Go compilers, such as gccgo [18] and gollvm [21], provide the necessary functionality. For this reason, we use gccgo to evaluate Go programs (§ 4). The key takeaway is that developers must carefuly

select a compiler toolchain that supports relocation, otherwise BFDOs may fail to take effect without any explicit warning.

3.3 Optimization

The optimization component optimizes the target binaries and produces optimized ones by leveraging profiles in two steps: 1) Profiling: collect profiles during execution using performance tools like `perf` [16] to sample the CPU-cycles event and taken branch stack. To minimize potential noise and ensure more accurate profiles, essential measures should be taken, including terminating irrelevant processes, increasing process priority, and setting CPU affinity. 2) Optimization: the binary is optimized using the profiles and options specified by the user.

During our experiments, we observed that the taken branch stack is arguably the most critical information for BFDOs, as it records a series of consecutive taken branches, thus providing accurate counts for critical edges to improve the resilience against imprecise sampling [36] (see § 4). Therefore, it is essential to enable this hardware feature if it is supported by the underlying hardware.

3.4 Quantitative Measurement

The measurement component calculates metrics for both the original and optimized binaries. Key metrics essential for measuring BFDOs include, but are not limited to, the following:

CPU cycles. This metric represents the number of clock cycles spent executing the program and is the primary target for reduction through binary optimization.

Branches and branch misses. This metric reflects the effect of branch prediction after basic block reordering, which can be reduced by effective block reordering.

Cache and cache misses. This metric encompasses both instruction cache (I-cache) and data cache (D-cache). BFDOs aim to improve instruction locality by reducing I-cache occupation, grouping hotspot code, and folding identical code. However, a trade-off between I-cache and D-cache optimization is often necessary, as optimizing both simultaneously can be challenging.

TLB and TLB misses. This metric indicates instruction and data locality at the page level, which is larger than cache size, and thus is useful for analyzing large programs.

These metrics can be measured by utilizing performance tools like `perf`. Meanwhile, the measurement should be repeated multiple times to minimize noise, before calculating the average speedups according to equation (1).

$$speedup = \frac{\sum_{repeat} \frac{unopt - opt}{unopt}}{repeat} \tag{1}$$

3.5 Root Cause Analysis

The root cause analysis component identifies the underlying causes of optimization failures, proposes suggestions, and provides fixes. We conduct manual analysis when automated analysis is impossible, following prior studies on optimization failures and performance degradation [3] [25] [45] [13] [49] [24]. Furthermore, we identify impactful factors for performance improvements, and provide best practices and suggestions to enhance the whole ecosystem.

4 Evaluation

In conducting the evaluation, we aim to answer the following research questions:
RQ1: Programming language diversity. Is BFDO effective on binaries compiled from different programming languages? What are the performance discrepancies for these binaries from different languages, if any?
RQ2: Architecture independence. Is BFDO dependent on different architectures, as these architectures have different hardware performance facilities?
RQ3: Optimization neutrality. How do compiler and linker optimizations affect BFDO? What are the optimization traps and pitfalls?

4.1 Experimental Setup

We conduct experiments on three current representative architectures: x86-64, AArch64, and RISC-V, as shown in Table 1. For Intel CPUs, all experiments are performed on Performance-cores (P-cores). Additionally, for RISC-V CPUs, we use Debian instead of Ubuntu because our RISC-V machine only provides the Debian images, which does not introduce any bias because our experiments do not depend on specific OS features. We use BOLT to optimize their I-cache locality with the following options:

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

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

4.2 Datasets

We construct two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world workloads, which are distributed in our open source.
Micro-benchmarks. We create a set of micro-benchmarks as a unified standard for four languages: C++, Go, Rust, and Standard ML (SML), which are compiled by clang, rustc, gccgo, and MLton, respectively.
Real-world workloads. We select diverse real-world open-source projects from GitHub based on the following selection criteria: 1) compiler support, and 2) performance priority. First, we select projects that satisfy the compiler requirements outlined in § 3.2, otherwise BFDO cannot be applied. Second, we select projects prioritizing performance, for which BFDOs are of more practical implications.

Table 1. Experimental setup for our evaluation.

	Hardware	OS	Compiler
x86-64	Intel i7-12700K 128GB RAM	Ubuntu 22.04	clang 18.1.1, rustc 1.77.1 gccgo 13.2.0, MLton 20231123
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
RISC-V	TH1520 SOC (4x Xuantie C910) 16GB RAM	Debian 12	clang 18.0.0, rustc 1.70.0 gccgo 13.2.0

Table 2. Performance for micro-benchmarks (% speedup, 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 symbol × means relocation error.

4.3 RQ1: Programming Language Diversity

To answer **RQ1** by demonstrating the effectiveness and efficiency of BFDO across multiple languages, we first conduct experiments on micro-benchmarks, as shown in Table 2. For all three architectures, C/C++ demonstrates speedups of 1.82%, 1.76%, and -5.69% on average, respectively, Go shows speedups of 3.38%, 2.63%, and -6.60% on average, respectively, whereas SML shows speedups of 3.4%, 1.23%, and 1.25% on average, respectively. While Rust exhibits the best optimization effects of 15.26% on x86-64, it performs poorly on RISC-V. In contrast, SML exhibits the worst optimization results on x86-64 and RISC-V but also the lowest variance.

Next, we evaluate real-world workloads, as presented in Table 3. Across all three architectures, the performance improvement for C/C++ achieves 4.17%, and 4.07% on average, while Go achieves 4.74% and 3.35%, followed by Rust's

Table 3. Performance for real-world workloads (% speedup, per test case).

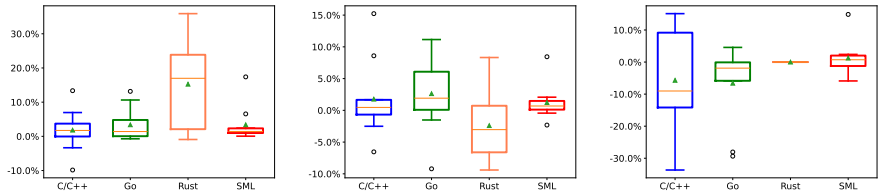
Language	Workload	x86-64	AArch64	RISC-V
C/C++	John	0.08	-0.46	3.87
	QuickJS	1.36	-0.28	⊗
	Redis	11.35	3.56	⊗
	Cppcheck	3.88	13.46	×
	Average	4.17	4.07	—
Go	gjson	5.58	9.28	-1.61
	goquery	1.24	1.08	-3.71
	websocket	1.69	1.46	18.41
	web framework	7.97	5.43	⊗
	grpc-go	7.24	-0.57	⊗
	Average	4.74	3.35	—
Rust	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.

4.36%. These results demonstrate that BFDO is effective in uniformly improving the performance of real-world workloads for all three languages.

Root cause analysis. We then investigate the root cause leading to performance discrepancies and identify the following factors. First, Rust and Go contain *more cold code* because they integrate more complex runtimes than C/C++, including stack overflow check, array bounds check, Drop trait code, and support libraries. Such mechanisms affect instruction locality hence benefit BFDOs. We identify that in our micro-benchmark, the cold code of C/C++ programs accounts for 57% of the code, while Go and Rust account for 84% and 96%, respectively, which results in section sizes 23x and 122x larger than C/C++. These metrics indicate that Go and Rust have greater optimization potential.

The second root cause for performance discrepancies is the *machine code style* generated from different languages. We reveal that BFDO disassembles binaries at the function granularity determined by the granularity of relocation information, which does not fit well with functional languages that often leverage continuation-passing style (CPS) transformation leading to many small functions (*i.e.*, chunks). Furthermore, we observe that the MLton compiler invokes the generated functions via indirect function pointer dispatches rather than direct calls, which complicates both the function disassembly and the final control-flow graphs. To this end, existing BFDOs still face grand challenges for functional languages.



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

Fig. 3. Box plots of micro-benchmark performance.

```
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:   00000013   nop
```

Fig. 4. Custom instructions encoded by `.insn` directive on RISC-V lead to optimization failures.

4.4 RQ2: Architecture Independence

To answer **RQ2**, we investigate BFDO's dependency on diverse architectures, as presented in Table 2. We observe that x86-64 improves the performance of all four languages, while AArch64 improves three among them except for Rust. RISC-V is the worst to manifest performance degradation for all languages, as depicted in Fig. 3. We then conduct experiments on real-world workloads, as detailed in Table 3. The results align closely with those from micro-benchmarks, confirming that x86-64 delivers the best performance, while RISC-V performs the worst. Furthermore, as Table 3 shows, AArch64 and RISC-V encounter more compilation, disassembly, and relocation errors, leading to optimization failures.

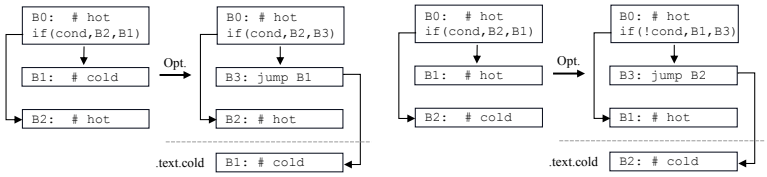
Root cause analysis. We then investigate the root causes of optimization failures and performance degradation, and identify four key root causes: 1) custom instructions, 2) relocation exception, 3) biased profiling, and 4) failed prediction. First, custom instructions in RISC-V lead to optimization failures. Specifically, as an open-source architecture, RISC-V allows chip vendors to incorporate custom instructions. While such customizations bring flexibility, they pose challenges to disassembly, an essential step for BFDO. Fig. 4 shows a failure example we observed when disassembling from `grpc-go`. The values `0x01a0000b` and `0x0246000b` in the `.insn` directive encode the custom instructions `SYNC.I` and `DCACHE.CVAL1` respectively on Xuantie C910 processor [8], which defeat the disassembler, leading to optimization failures.

```
foo:
  # expand by la a0, symbol
  auipc a0, %pcrel_hi(symbol)  # R_PCREL_HI20 (symbol)
  addi  a0, a0, %pcrel_lo(foo) # R_PCREL_LO12_I (foo)
bar:
  # expand by la.tls.gd a0, symbol
  auipc a0,0                   # R_TLS_GD_HI20 (symbol)
  addi  a0,a0,0                # R_PCREL_LO12_I (bar)
```

Fig. 5. PC-relative relocations lead to optimization failures.

The second root cause leading to optimization failures is *relocation exceptions*. Specifically, we identify that optimization failures for Rust on RISC-V are due to the exceptions that the optimizer failed to find the corresponding `%pcrel_hi` modifier. This exception stems from the relocation type *PC-relative TLS GD GOT reference* used in RISC-V multithreaded programs, which is used to load the address of a symbol into a register (via the pseudo directive `la` or `la.tls.gd` for thread-local storage (TLS)). On most architectures, such load operations typically use PC-relative addressing, which requires two instructions on RISC-V, as shown in Fig. 5. The `%pcrel_hi` modifier represents the high 20 bits of the relative address between the program counter (PC) and the target symbol, and the `%pcrel_lo` modifier represents the low 12 bits. In this example, to load the address of `symbol` into register `a0`, `auipc` instruction uses the high 20 bits of the relative address between PC and `symbol` as source operand HI, storing $PC + (HI << 12)$ into register `a0`. Then `addi` instruction takes register `a0` and the low 12 bits of the relative address between corresponding `auipc` instruction and `symbol` as source operands, adding them and storing the result in register `a0`. The instructions in `foo` and `bar`, though written differently, perform similar functions. They load normal symbols and TLS symbols, respectively, and produce two distinct relocation types: `R_PCREL_HI20` and `R_TLS_GD_HI20`. We identify that BFDO fails to resolve the `R_TLS_GD_HI20` relocation type, leading to optimization failures. Furthermore, we reveal that such relocation types are common in Rust binaries even for non-multithreaded programs but are rare in C/C++ and Go binaries. Thus we believe existing BFDOs have spaces to improve the relocation type resolution in Rust binaries.

The third root cause leading to performance degradation is *biased profiles*. Specifically, while sample-based profiling is effective for generating profiles without any special aid from instrumentation, it may not reflect the real execution frequency of the program's basic blocks. This inaccuracy stems from factors such as sampling frequency, basic block size, and sampling environment. We observe that among all these factors, the basic block size is the most critical yet challenging factor to tune by adjusting sampling options. Furthermore, sample-based profiling may introduce bias because it is usually performed at OS interrupt granularity, in which larger basic blocks with more instructions have a larger sampling probability while smaller basic blocks are rarely sam-



(a) Case 1 (b) Case 2

Fig. 7. Failed branch predictions lead to optimization failures.

pled. Although increasing the sampling frequency can improve the sampling probability of small basic blocks, it simultaneously increases the sampling probability of large basic blocks, making it difficult to achieve balanced profiling. We present in Fig. 6 an example leading to optimization failures due to biased profiles. Basic block `B1`, `B3`, and `B4` have the same execution count. However, basic block `B3` is rarely sampled due to its small size (one instruction). Consequently, BFDOs may classify `B1` and `B4` as hot basic blocks while `B3` are classified as cold block based on the sampling results. Thus the optimization leverages the hot/cold split algorithm to split them into distinct sections that are far apart in address space, causing each loop to jump twice between the hot and cold sections. Such extra jumps introduced by the optimization lead to severe performance degradation of I-cache locality, and in the worst case lead to a 1.5x increase in execution time.

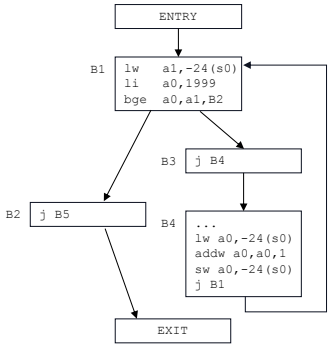


Fig. 6. Biased profiles lead to optimization failures.

Failed branch prediction is the fourth and last root cause we have identified. During BFDO, the optimizer reorders basic blocks by adjusting the target of branch instructions, placing hot basic blocks in fallthrough areas and cold basic blocks in cold sections. However, the reordered basic blocks may be too far away and beyond the addressing range (typically ±4KB) of the conditional jump instruction that reaches them, as shown in Table 4. To address this issue, the optimizer can insert an unconditional jump instruction following the conditional jump, but this approach may incur significant branch prediction failures. We present two failed branch predictions leading to optimization failures in Fig. 7. In the first case, 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 placed into the cold section, and an unconditional jump instruction to `B1` is inserted in the original position of `B1`. However, while such a code layout improves code locality, it does not improve branch prediction accuracy. In the second case, the situation is even worse. To jump to cold basic

block `B2`, the optimizer reverses the condition of the conditional jump instruction and inserts an unconditional jump before basic block `B1`, incurring a branch prediction failure for each jump to `B1`. We observe that when the improvement from code locality cannot compensate for the overhead of branch prediction failures, the overall performance will degrade. Specifically, we identify that branch prediction failures in if statements incur a performance degradation of about 6%.

Table 4. Max jump range on three architectures.

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

4.5 Optimization Neutrality

To answer **RQ3** by investigating optimization neutrality of BFDO, we conduct experiments by applying compiler and linker optimizations when generating the binaries.

We first experiment with different compiler optimization options, and reveal that lower optimization levels (e.g., `O0`) will degrade performance because they tend to generate more complex control-flow graphs with numerous tiny basic blocks. For instance, we observe a 4.0% performance improvement when running C/C++ micro-benchmarks on RISC-V by simply switching the compiler optimization option from `O0` to `O2`. Therefore, we argue that full compiler optimizations should be utilized to maximize the benefits of BFDO.

We then conduct experiments to test the effects of link-time optimizations on BFDOs. We reveal that BFDO often assumes symbol locations remain unchanged after linking, but binary optimization breaks that assumption and thus may lead to optimization failures. Specifically, we observe that the Cppcheck workload encounters the relocation exception on RISC-V where `R_RISCV_JAL` relocation exceeds its address range. Upon investigating the source code, we identify that this exception is triggered by the linker relaxation optimization. The linker relaxation optimization is a link-time optimization aimed at reducing code size. Unfortunately, the binary optimizations in BFDO, including function reordering, basic block reordering, and the hot/cold code split, will change the symbol locations, triggering exceptions. To investigate mitigations, we recompile the Cppcheck workload with the `--no-relax` option to disable linker relaxation optimization, and observe a performance improvement of 21.02%.

5 Best Practices and Future Directions

In this section, we present best practices for using BFDOs and outline directions for future studies.

Better support of relocation information. We argue that a key step to utilize BFDOs is selecting compilers that support the export of relocation information. Specifically, since BFDOs require precise disassembly and relocation information, compilers should export symbol table and precise relocation information to binary files. While many compilers support symbol table well, some do not support relocation information (*e.g.*, the `gc` compiler for Go). Besides, optimizer support for certain relocation types, such as `R_RISCV_TLS_GD_HI20`, remains limited. Therefore, using unsupported relocation types should be avoided. This is a trade-off between performance and program implementation.

Do not use BFDOs on unoptimized binaries. We argue that BFDOs supplement existing compilation optimizations but do not replace them. We observe from our evaluation that unoptimized binaries (e.g., generated from the compiler via `O0`) often contain numerous tiny basic blocks and complex jump structures, which are more susceptible to biased profiles, especially on RISC architectures like RISC-V. Applying BFDOs on such binaries often leads to performance degradation rather than improvements.

Avoid using linker relaxation. We argue that although linker relaxation can reduce binary size and improve performance, it has negative impacts on BFDOs, especially for large binaries. To this end, we suggest that linker relaxation should be avoided during BFDOs. In the long term, we believe that the conflict between linker relaxation and BFDOs should be addressed by introducing more sophisticated optimization heuristics that take into account the effects of BFDOs on code layouts. We identify that this idea is integrated into the recent version of BOLT, but regrettably, it does not work correctly, at least as of this study.

Improve sampling accuracy. We argue that sampling accuracy is essential for BFDOs and thus should be carefully tuned. We reveal from our experiments that it is often effective to adjust the sampling frequency to improve accuracy (*e.g.*, the `-F` option in `perf`), when the performance improvement is suboptimal. Furthermore, due to the imprecision of sample-based profiling for tiny basic blocks, we argue that sample-based profiling can be complemented by instrumentation-based profiling to determine more precise profiles. We believe this best practice is especially valuable for hardware that lacks powerful performance measurement units.

Better disassembly and hardware support. We argue that effective BFDOs comprise two key sub-tasks: binary disassembly and optimization. Unfortunately, both sub-tasks are well-known NP-complete problems and thus effective heuristics must be leveraged. To this end, we believe better disassembly support will benefit BFDOs, especially for RISC binaries in which the mixture of code and data often defeats disassembly. Furthermore, we observe that current architectures, especially AArch64 and RISC-V, still have limited hardware profiling support, which undermines BFDOs. Addressing this limitation requires attention and effort from hardware vendors.

6 Discussion and Threats to Validity

Like any empirical study, there are threats to the validity of our results. We discuss the most significant ones, along with possible mitigations.

External validity. Threats to external validity concern the generalizability of findings across experimental settings, primarily stemming from three factors: the selected programming languages, architectures, and benchmarks. To mitigate these threats, we selected four programming languages spanning distinct application domains and encompassing varied programming paradigms. Additionally, we evaluated three representative architectures. Finally, for benchmarks, we used both micro-benchmarks and real-world workloads applied across various fields.

Internal validity. As for threats to internal validity, a critical threat stems from random variance in execution times of benchmarks, caused by subtle fluctuations in the host OS's state that may manifest as significant deviations. To mitigate this threat, we conducted multiple trials for each benchmark, using the mean execution duration as the evaluation metric while calculating temporal variance to quantify runtime volatility. For micro-benchmarks whose execution speed exceeds measurement resolution thresholds, we incorporated repeated invocations to amplify temporal observables, thereby compensating for sampling inaccuracies.

Construct validity. We evaluated the effectiveness of BFDOs through comparative analysis of runtime data collected from benchmarks before and after optimization. Although many benchmarks on RISC-V platforms were non-executable due to compilation errors or optimization failures, these instances serve as critical validation indicators for assessing the cross-platform validity of BFDOs.

7 Related Work

Binary optimizations. There are numerous studies on binary optimizations. Panchenko et al. [36] presented a static binary optimizer BOLT for data-center applications, demonstrating that profile data can be used more precisely at the binary level. Williams-King and Yang et al. [48] implemented CodeMason, which performs static binary rewriting based on the binary rewriting platform called Egalito [47]. Panchenko et al. [37] demonstrated Lightning BOLT 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. [52] presented a framework called Janus to address the challenge of automatic binary parallelization. Savage and Jones et al. [39] proposed a post-link profile-guided optimization tool called HALO to improve the layout of heap data to reduce cache misses automatically.

However, existing work mainly focuses on the x86 CPU architecture and binaries compiled from C/C++. Our work studies the effectiveness and performance improvement of other platforms and programming languages, aiming to fill this gap.

Profiling techniques. Profiling techniques are a key component of binary optimization. Instrumentation-based profiling is a common method for collecting

precise profile data, but it often incurs non-negligible runtime overhead. To mitigate this issue, Ball et al. [6] optimized probe placement using minimal spanning tree. Additionally, Cho et al. [23] proposed a novel instrumentation framework that reported an average runtime slowdown of 3% to 6%. Since the overhead of instrumentation-based profiling remains undesirable for production deployment, sample-based profiling is widely adopted by many tools. He et al. [22] proposed a pseudo-instrumentation technique as a supplement to sample-based profiling to improve profile quality without incurring the overhead of traditional instrumentation. Moreira et al. [33] enhanced static profiling technique called Evidence-Based Static Prediction (ESP), originally proposed by Calder et al. [7], which enables the generated profiles to be used for binary optimization.

However, although leveraging profiling techniques as foundational infrastructure, our contribution lies in systematically evaluating BFDOs' effectiveness and efficiency. This focus dictates our selection of the most compatible and relatively accurate sample-based profiling in our study.

8 Conclusion

In this work, we present the first study on binary feedback-directed optimization, aiming to evaluate its current capabilities, remaining limitations, and future research directions. By designing and implementing a software prototype, we investigate the effectiveness and efficiency of binary optimization across three representative architectures and four programming languages. We reveal three root causes of optimization failures and performance degradation, along with three best practices for improved optimization outcomes. Additionally, we outline three promising directions for future research. We believe that our study provides valuable insights that can guide future research efforts in leveraging modern hardware performance measurement capabilities through advanced compiler technologies.

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. ACM SIGPLAN Notices **32**(5), 85–96 (May 1997)
2. Anderson, J.M., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A., Weihl, W.E.: Continuous profiling: Where have all the cycles gone? ACM Transactions on Computer Systems **15**(4), 357–390 (Nov 1997)
3. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An {In-Depth} analysis of disassembly on {Full-Scale} x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 583–600 (2016)
4. Appel, A.W.: Compiling with Continuations. Cambridge University Press, USA (2007)
5. ARM: Embedded trace macrocell architecture specification (2024), https://developer.arm.com/documentation/ihi0014/q/Introduction/About-Embedded-Trace-Macrocells

6. Ball, T., Larus, J.R.: Optimally profiling and tracing programs. ACM Transactions on Programming Languages and Systems **16**(4), 1319–1360 (Jul 1994)
7. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based static branch prediction using machine learning. ACM Transactions on Programming Languages and Systems **19**(1), 188–222 (Jan 1997)
8. Chen, C., Xiang, X., Liu, C., Shang, Y., Guo, R., Liu, D., Lu, Y., Hao, Z., Luo, J., Chen, Z., Li, C., Pu, Y., Meng, J., Yan, X., Xie, Y., Qi, X.: 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). pp. 52–64. IEEE, Valencia, Spain (may 2020)
9. Chen, D., Li, D.X., Moseley, T.: Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 12–23. ACM, Barcelona Spain (Feb 2016)
10. Chen, D., Vachharajani, N., Hundt, R., Li, X., Eranian, S., Chen, W., Zheng, W.: Taming hardware event samples for precise and versatile feedback directed optimizations. IEEE Transactions on Computers **62**(2), 376–389 (Feb 2013)
11. Chen, S., Wang, H.: Compiler support for linker relaxation in risc-v (2019), https://riscv.org/wp-content/uploads/2019/03/11.15-Shiva-Chen-Compiler-Support-For-Linker-Relaxation-in-RISC-V-2019-03-13.pdf
12. Cohn, R., Goodwin, D., Lowney, P.G., Rubin, N.: Spike: an optimizer for alpha/nt executables. In: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997. p. 3. NT'97, USENIX Association, USA (1997)
13. Committee, P.I.: Tech-privileged@lists.riscv.org | a proposal to enhance risc-v hpm (hardware performance monitor) (2024), https://lists.riscv.org/g/tech-privileged/topic/a_proposal_to_enhance_risc_v/75675990
14. Conte, T.M., Patel, B.A., Menezes, K.N., Cox, J.S.: Hardware-based profiling: An effective technique for profile-driven optimization. International Journal of Parallel Programming **24**(2), 187–206 (Apr 1996)
15. Corporation, I.: Intel® 64 and ia-32 architectures software developer's manual (325384-083US) (Mar 2024)
16. De Melo, A.C.: The new linux'perf'tools. In: Slides from Linux Kongress. vol. 18, pp. 1–42 (2010)
17. Domingos, J.M., Tomas, P., Sousa, L.: Supporting risc-v performance counters through performance analysis tools for linux (perf) (Dec 2021)
18. Free Software Foundation, I.: Top (the gnu go compiler) (2024), https://gcc.gnu.org/onlinedocs/gccgo/
19. Fried, J., Chaudhry, G.I., Saurez, E., Choukse, E., Goiri, I.n., Elnikety, S., Fonseca, R., Belay, A.: Making kernel bypass practical for the cloud with junction. In: Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation. NSDI'24, USENIX Association, USA (2024)
20. Gloy, N., Wang, Z., Zhang, C., Chen, B., Smith, M.: Profile-based optimization with statistical profiles (1997)
21. Google: Gollvm - git at google (2024), https://go.googlesource.com/gollvm/
22. He, W., Yu, H., Wang, L., Oh, T.: Revamping sampling-based pgo with context-sensitivity and pseudo-instrumentation. In: 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 322–333 (Mar 2024)

23. Hyoun Kyu Cho, Moseley, T., Hank, R., Bruening, D., Mahlke, S.: Instant profiling: Instrumentation sampling for profiling datacenter applications. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–10. IEEE, Shenzhen (Feb 2013)
24. Ibáñez, R.F.: 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 (2024), https://github.com/riscv-non-isa/riscv-elf-psabi-doc/issues/90
25. Jiang, M., Zhou, Y., Luo, X., Wang, R., Liu, Y., Ren, K.: An empirical study on arm disassembly tools. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 401–414. ISSTA 2020, Association for Computing Machinery, New York, NY, USA (Jul 2020)
26. Kennedy, A.: Compiling with continuations, continued. SIGPLAN Not. **42**(9), 177–190 (Oct 2007)
27. Knuth, D.E., Stevenson, F.R.: Optimal measurement points for program frequency counts. BIT **13**(3), 313–322 (Sep 1973)
28. for Linux Team, R.: Rust for linux (2024), https://rust-for-linux.com
29. LLVM: Llvm support for bolt (2024), https://github.com/llvm/llvm-project/tree/main/bolt
30. LLVM: Optimizing clang with llvm bolt (2024), https://github.com/llvm/llvm-project/blob/main/bolt/docs/OptimizingClang.md
31. LLVM: [samplefdo] flow sensitive sample fdo (fsafdo) profile loader (2024), https://reviews.llvm.org/D107878
32. Luk, C.K., Muth, R., Patil, H., Cohn, R., Lowney, G.: 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. pp. 15–26 (Mar 2004)
33. Moreira, A.A., Ottoni, G., Quintão Pereira, F.M.: Vespa: Static profiling for binary optimization. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 144:1–144:28 (Oct 2021)
34. Mozilla: Language details of the firefox repo (2024), https://4e6.github.io/firefox-lang-stats/
35. Novillo, D.: Samplepgo - the power of profile guided optimizations without the usability burden. In: 2014 LLVM Compiler Infrastructure in HPC. pp. 22–28. IEEE, LA, USA (Nov 2014)
36. Panchenko, M., Auler, R., Nell, B., Ottoni, G.: Bolt: A practical binary optimizer for data centers and beyond. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14. IEEE, Washington, DC, USA (Feb 2019)
37. Panchenko, M., Auler, R., Sakka, L., Ottoni, G.: Lightning bolt: Powerful, fast, and scalable binary optimization. In: Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction. pp. 119–130. ACM, Virtual Republic of Korea (Mar 2021)
38. Pike, R.: New case studies about google's use of go (2024), https://opensource.googleblog.com/2020/08/new-case-studies-about-googles-use-of-go.html
39. Savage, J., Jones, T.M.: Halo: Post-link heap-layout optimisation. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. pp. 94–106. CGO 2020, Association for Computing Machinery, New York, NY, USA (Feb 2020)
40. Shen, H., Pszeniczny, K., Lavaee, R., Kumar, S., Tallam, S., Li, X.D.: 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. pp. 617–631. ACM, Vancouver BC Canada (Jan 2023)
41. Srivastava, A., Eustace, A.: Atom: A system for building customized program analysis tools. ACM SIGPLAN Notices **29**(6), 196–205 (Jun 1994)
42. Suzaki, K., Nakajima, K., Oi, T., Tsukamoto, A.: Ts-perf: General performance measurement of trusted execution environment and rich execution environment on intel sgx, arm trustzone, and risc-v keystone. IEEE Access **9**, 133520–133530 (2021)
43. TIOBE: Tiobe index (2024), https://www.tiobe.com/tiobe-index/
44. Wang, R., Gibson, D., Rodrigues, K., Luo, Y., Zhang, Y., Wang, K., Fu, Y., Chen, T., Yuan, D.: µslope: high compression and fast search on semi-structured logs. In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation. OSDI'24, USENIX Association, USA (2024)
45. Wang, S., Wang, P., Wu, D.: Reassembleable disassembling. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 627–642 (2015)
46. Waterman, A., Asanovic, K., Hauser, J., Inc., S.: The risc-v instruction set manual volume ii: Privileged architecture document version 20211203. CS Division,EECS Department, University of California, Berkeley (Dec 2021)
47. Williams-King, D., Kobayashi, H., Williams-King, K., Patterson, G., Spano, F., Wu, Y.J., Yang, J., Kemerlis, V.P.: Egalito: Layout-agnostic binary recompilation. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 133–147. ASPLOS '20, Association for Computing Machinery, New York, NY, USA (Mar 2020)
48. Williams-King, D., Yang, J.: Codemason: Binary-level profile-guided optimization. In: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation. pp. 47–53. FEAST'19, Association for Computing Machinery, New York, NY, USA (Nov 2019)
49. Yan, L., Thompson, D., Support, L.: Using perf on arm platforms (2024), https://static.linaro.org/connect/yvr18/presentations/yvr18-416.pdf
50. Yu, L., Zhang, X., Zhang, H., Sonchack, J., Ports, D., Liu, V.: Beaver: practical partial snapshots for distributed cloud services. In: Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation. OSDI'24, USENIX Association, USA (2024)
51. Yuan, P., Guo, Y., Chen, X.: Experiences in profile-guided operating system kernel optimization. In: Proceedings of 5th Asia-Pacific Workshop on Systems. pp. 1–6. ACM, Beijing China (Jun 2014)
52. Zhou, R., Jones, T.M.: Janus: Statically-driven and profile-guided automatic dynamic binary parallelisation. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 15–25 (Feb 2019)