Empirical Observations About Profile-Guided Optimizations for Mainstream C/C++ Compilers

Soma Pal
soma_pg2021@ku.edu
University of Kansas,
USA

Prasad A. Kulkarni
prasadk@ku.edu
University of Kansas,
USA

Abstract

The idea behind profile-guided optimizations (PGO) is to monitor different aspects of a program's run-time behavior, and then employ this information to guide decisions during individual compiler optimizations to improve program performance. PGO is a mature technology that is available in most mainstream compilers and is widely regarded to benefit performance. In this work, we conduct the first comprehensive empirical study of the behavior and properties of PGOs in two state-of-the-art mainstream C/C++ compilers, GCC and Clang, evaluated using MiBench embedded system benchmarks on x86-64 platform. Our study reveals many interesting, some expected and some counter-intuitive observations about PGOs in mainstream C/C++ compilers. We believe our intellectually intriguing observations will help compiler designers and software developers further develop and usefully deploy this technology.

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

Keywords: Profile-guided optimizations, compilers, performance

ACM Reference Format:
Soma Pal and Prasad A. Kulkarni. 2026. Empirical Observations About Profile-Guided Optimizations for Mainstream C/C++ Compilers. In *Proceedings of the 27th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '26), June 15–16, 2026, Boulder, CO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3814943.3816180

1 Introduction

High-level programming languages allow developers to write code at a higher level of abstraction that is cleaner, more concise, more intuitive to read, and easier to extend, update and maintain later. Often, the compiler is then used to translate this high-level code into an efficient and correct binary. Compilers employ many specialized algorithms, called *optimizations*, that are tasked with the responsibility to transform and improve the structure, layout and composition of the generated code into a semantically-equivalent variant that can achieve good run-time performance (typically, speed) on the underlying computer hardware.

Several compiler optimizations, especially in ahead-of-time compilers, like those for C/C++, employ static-time *heuristics* to guide their application. For instance, the compiler may use static heuristics to decide the best basic-block layout for a function, or determine the registers to spill during register allocation, or choose the methods to inline during function inlining. These static heuristics in mature compilers are carefully tuned by experienced compiler writers for the common case across many programs. Yet, these heuristics may still cause the compiler to generate code with poor performance characteristics, especially when they do not correctly model the actual run-time behavior of the program.

Profile-guided optimizations (PGO) is an alternate technology that employs the run-time program behavior information, rather than or in addition to static program analysis, to guide compiler decisions [43]. Program profiling in ahead-of-time (AOT) compilers typically uses prior runs of the program to collect information about the execution-time behavior of the program. Profile information can be collected using binary *instrumentation* or *sampling*. The application of PGOs in AOT instrumentation-based compilers typically operates in three steps: (a) Firstly, the compiler instruments the binary by inserting code to monitor aspects of program execution-time behavior, (b) Secondly, the instrumented binary is executed with some *representative* program inputs, which causes the instrumentation code to output profile data at run-time. (c) Lastly, the compiler uses the profile data in a subsequent compile of the program to fine-tune the optimizations and regenerate the program binary.

Profiling and PGOs represent a mature technology with support in most production-grade C/C++ compilers. Yet, the benefits and properties of PGOs in *mainstream* C/C++ compilers, including the popular GCC [23] and LLVM [16] compilers, have not been comprehensively explored in recent years and remain poorly understood. For instance, while the benefits developers attain from PGOs for their projects for some typical use-cases are often reported [51], detailed

LCTES '26, Boulder, CO, USA

ACM ISBN 979-8-4007-2721-4/2026/06
https://doi.org/10.1145/3814943.3816180

analysis regarding behavior of PGOs, best and worst case PGO results, etc. are typically unavailable. Other properties of PGOs, such as the impact of ideal case and cumulative profile data on PGO performance, and their sensitivity to profile data with regards to code generation and performance, are also unknown. Given the maturity of the field, one of our aims is to also investigate if PGO properties across contemporary compilers reveal similar results and trends, or if there is still an opportunity for compilers to learn from each other to mutually benefit and improve.

Our goals for this work are, through a series of well-designed experiments and with multiple mature and contemporary C/C++ compilers (GCC and Clang, in this study) (a) to explore important properties and evaluate the performance impact of instrumentation-based offline PGOs, on the embedded systems benchmark suite, MiBench, running on an x64 hardware platform, and (b) to compare and understand if and how the properties of PGOs correlate across compilers.

Some important contributions from this work are:

1. Our experiments show that while PGOs can help improve program performance significantly for some program inputs, they can also cause large losses, in adverse cases.
2. Our experiments reveal that the *ideal* profile does not always produce the best performance from PGOs.
3. We find that profile data aggregation done for the *cumulative* profile often lowers performance over the best profile.
4. We find that the benefit individual optimizations realize from profile data is uneven and unpredictable.
5. We find poor correlation between most PGO properties, including in the level of benefit achieved, across the two mainstream compilers employed for this work.

The rest of the paper is organized as follows. We present related works in Section 2. We present details of our experimental setup in Section 3. We explain the experimental results in Section 4. We present directions for future work in Section 5, and our conclusions in Section 6.

2 Background and Related Works

The term *profile data* represents any quantitative information about the execution time behavior of a program [32, 40]. Examples include, basic block or instruction execution frequency counts, function invocation and call-site counts, conditional branch direction and program path profiles, and data object reference counts. Compiler writers may employ this profile data to guide certain decisions during optimizations, such as register allocation to select the variables to promote to registers [41, 46], basic block layout to improve cache behavior [37], instruction scheduling to reduce hazards and improve instruction-level parallelism [12, 50], and to balance varying trade-offs during function inlining [5, 9, 10], loop unrolling [45], and selective compilation [3].

Profiling data can be collected in two ways: offline and online [43]. Offline profiling uses prior program runs to collect profile data, and later compilation uses that profile information for optimization decisions. In general, static or AOT compilers, like GCC/g++ use this multi-step offline profiling approach [11, 38, 49]. Dynamic or online profiling collects profile data during each execution, and is commonly used by just-in-time compilers in advanced managed languages runtimes, like those for Java [4, 15, 31]. Our current work explores the behavior of PGOs in contemporary static C/C++ compilers employing offline profiling.

The *instrumentation*-based profiling technique is a prominent profiling strategy that inserts software counters in the binary code to record and collect detailed profile information, like block, function, and call-site execution counts, and how often branches are taken during execution [11, 24]. While this code instrumentation-based technique can produce accurate profiles, it suffers due to high overhead of the instrumented binary [20]. *Sampling*-based profiling uses OS or library support to periodically collect the program's execution information, such as the current instruction address, function, or call stack, to determine the code executing at that time. Over time, the collected profile information is aggregated to determine the hot blocks, or methods, or paths, etc. While instrumentation-based technique can collect more precise profile information more quickly, the sampling-based approach has lower overhead and is less intrusive, making it more suitable for production environments [26, 28].

Researchers have suggested other strategies to reduce the profiling overhead, most of which then yield imprecise, approximate or incomplete program information. Some techniques switch between the instrumented and non-instrumented code at run-time [6, 33]. Other techniques attempt to lower overhead by using low-level hardware performance counters [1, 17]. Hardware-assisted sampling based techniques sample the value of the instruction pointer on different aspects of the execution-time program behavior [18, 48]. Such low-overhead profiling systems can be employed in production systems to continuously monitor application behavior after deployment [2, 13, 39]. Other techniques help correlate the stale or approximate hardware samples with high-level source code [13, 29]. In this work we only use the offline software instrumentation based profiling systems that are the default mode for mainstream C/C++ compilers.

The offline profiling technique generates a single immutable binary customized to the provided profile data. A binary that is customized using PGOs guided by the provided profile data may deliver poor performances for other different or non-representative program inputs. Therefore, a major challenge to successfully employing the offline profiling approach for PGOs is finding program inputs that are representative of the overall program execution behavior. Researchers studied the benefit of representative profiles on PGOs and found that *real* run-time profiles are better than statically derived profiles

Table 1. Properties of MiBench benchmarks with MiDatasets

Benchmark	Exec. Func.	Total Blocks	Benchmark	Exec. Funcs.	Total Blocks
automotive_bitcount	19	101	office_ghostscript	3272	39012
automotive_qsort1	4	61	office_rsynth	41	754
automotive_susan_c	20	702	office_stringsearch1	11	190
automotive_susan_e	20	702	security_blowfish_d	8	136
automotive_susan_s	20	702	security_blowfish_e	8	136
consumer_jpeg_c	291	4234	security_pgp_d	293	5393
consumer_jpeg_d	274	3944	security_pgp_e	293	5856
consumer_lame	192	4034	security_rijndael_d	8	162
consumer_tiff2bw	303	6489	security_rijndael_e	8	162
consumer_tiff2rgba	305	6476	security_sha	8	52
consumer_tiffdither	303	6471	telecom_CRC32	5	25
consumer_tiffmedian	306	6647	telecom_adpcm_c	3	66
network_patricia	6	152	telecom_adpcm_d	3	66
			telecom_gsm	63	707

but not as good as perfect profiles from the same run [47]. Various approaches have also been used to study and statistically combine the profiles over multiple runs to account for the variations in execution behavior across inputs [8, 19, 42].

To our knowledge, there are no recent works that conduct a thorough investigation and comparison of the PGO systems employed by contemporary AOT compilers. Modern compilers, conceivably, leverage the vast prior knowledge regarding profiling mechanisms and PGOs to implement their systems. Our goal in this work is to conduct a systematic exploration of many important properties of PGOs, including several that have never been studied, in the context of mainstream C/C++ compilers. We also study how PGO properties correlate and vary across mature compilers.

3 Experimental Setup

In this section we describe our benchmarks, machine configuration, and experimental design employed for this work.

3.1 Benchmarks, Machine & Compiler

We use the MiDataSets [21] suite for this work. MiDataSets extends MiBench [25], which is a suite of benchmarks intended to represent commercial embedded systems use cases. In particular, MiDataSets expands the number of inputs for each benchmark to 20 that are collected from various *sources* and varying *size*, *nature*, and data set *properties* [21]. The data sets were collected to study the influence of program input on the performance benefit obtained from compiler optimizations, in general, and iterative compilation, in particular, and hence seemed a good fit for this work on PGOs.

MiDataSets includes minor compatibility fixes to MiBench benchmarks for 64-bit architectures. However, we found that the `ispell` benchmark doesn't support x86_64. We also could not use `mad` and `dijkstra`, as these two benchmarks require a library we could not build. These three benchmarks are excluded from our experiments and analysis, leaving us with 27 benchmarks. Table 1 lists the benchmarks used in our study along with the count of union of functions executed during a benchmark run with *any* of their inputs and the sum of the number of basic blocks in those functions.

Table 2. Flags to Generate Profile Data and Enable PGOs

	Instrument Binary to Generate Profile Data	Use Profile Data and Enable PGOs
Clang	-fprofile-instr-generate (-fprofile-generate)	-fprofile-instr-use (-fprofile-use)
GCC	-fprofile-generate	-fprofile-use

All our experiments and measurements are performed on a server with an Intel(R) Xeon(R) CPU E5-2660 @2.20GHz processor and 16 GB of RAM. The server's operating system is Linux. We use the CPU cycle count metric provided by the Linux performance analysis tool, called *perf* or *perf_events*, to collect program performance measurements.

We use two of the most popular, state-of-the-art and contemporary C/C++ compilers, GCC [23] (version 11.3.1) and LLVM/Clang [16] (version 15.0.0) for this work. We use the -O3 optimization level for our baseline performance.

Clang offers two profile collection flags, *-fprofile-instr-generate* that is documented as a front-end PGO flag, and *-fprofile-generate* mentioned as a middle-end PGO flag. We collected all our results with both these profiling flags, and found that their overall trends, patterns and observations relevant to this work are identical. Yet, using the *-fprofile-instr-generate* flag consistently provides slightly better performance results. Hence, we only present the results with this one profiling flag for Clang in this work. Table 2 lists the flags we employ to enable and use the instrumentation based profiling and PGO support provided by the compilers.

3.2 Experimental Data Collection

In this section we present our automated data collection setup and explain terms we employ in the rest of this paper.

We employ the appropriate flags with each compiler to generate the respective instrumented binaries for each of the 27 benchmarks that then collect profile information for all the 20 inputs provided by MiDataSets. Then, we generate PGO-enabled binaries for each benchmark by feeding the different profile data, which generates 20 distinct custom binaries for each program-input/profile pair. We use the tools provided by each compiler framework to combine the 20 profiles for each program into a *cumulative* profile. We use the cumulative profile to generate the 21^{st} customized binary for each benchmark. Finally, we compile each benchmark with the '-O3' flag with both compilers (22^{nd} baseline binary).

Next, for each program, we collect performance numbers (CPU cycles) by running the 21 customized binaries along with the baseline (O3) binary with each of the 20 inputs. We run each binary-input pair 12 times and compute the average and standard deviation, which results in 440 performance measures (22 binaries * 20 inputs) for each benchmark. For each program-input pair, we divide their average performance with the corresponding O3-binary performance to calculate the benefit delivered by PGOs in each scenario.

Table 3. Illustrative color-coded sample matrix of collected performance for 20 custom binaries (in rows) evaluated across 20 inputs (in columns) for a hypothetical benchmark and data: PGO-Best, PGO-Worst, PGO-Same, PGO-Same-Best, PGO-Same-Worst, Baseline (O3), PGO-Cumul, PGO-Cumul-Best, PGO-Cumul-Worst.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	0.85	0.84	0.85	0.83	0.84	0.83	0.84	0.84	0.83	0.84	0.87	0.89	0.87	0.85	0.85	0.84	0.84	0.83	0.84	0.84
2	0.86	0.88	0.87	0.88	0.87	0.87	0.83	0.86	0.86	0.85	0.86	0.88	0.85	0.89	0.84	0.84	0.86	0.86	0.84	0.83
3	0.86	0.88	0.87	0.88	0.87	0.87	0.83	0.86	0.86	0.85	0.86	0.88	0.85	0.89	0.84	0.84	0.86	0.86	0.84	0.83
4	0.84	0.86	0.85	0.86	0.85	0.86	0.82	0.85	0.84	0.83	0.84	0.87	0.83	0.87	0.83	0.81	0.84	0.84	0.82	0.82
5	0.84	0.86	0.85	0.86	0.85	0.86	0.82	0.85	0.84	0.83	0.84	0.87	0.83	0.87	0.83	0.81	0.84	0.84	0.82	0.82
6	0.86	0.88	0.87	0.88	0.87	0.87	0.83	0.86	0.86	0.85	0.86	0.88	0.85	0.89	0.84	0.84	0.86	0.86	0.84	0.83
7	0.87	0.93	0.90	0.93	0.92	0.91	0.82	0.87	0.88	0.86	0.91	0.90	0.88	0.91	0.87	0.84	0.89	0.88	0.82	0.82
8	0.87	0.90	0.87	0.90	0.88	0.89	0.87	0.88	0.89	0.85	0.88	0.92	0.86	0.92	0.86	0.84	0.87	0.88	0.88	0.86
9	0.88	0.92	0.92	0.93	0.93	0.92	0.85	0.88	0.89	0.86	0.89	0.93	0.88	0.93	0.86	0.85	0.89	0.89	0.85	0.84
10	0.87	0.93	0.90	0.93	0.92	0.91	0.82	0.87	0.88	0.86	0.91	0.90	0.88	0.91	0.87	0.84	0.89	0.88	0.82	0.82
11	0.87	0.90	0.87	0.90	0.88	0.89	0.86	0.88	0.89	0.86	0.88	0.92	0.86	0.92	0.86	0.84	0.88	0.88	0.87	0.85
12	0.87	0.94	0.89	0.93	0.90	0.91	0.78	0.87	0.90	0.86	0.92	0.89	0.93	0.87	0.84	0.90	0.89	0.83	0.82	0.81
13	0.85	0.90	0.88	0.90	0.89	0.89	0.81	0.85	0.86	0.84	0.88	0.88	0.86	0.88	0.84	0.82	0.87	0.86	0.81	0.81
14	0.87	0.94	0.89	0.93	0.90	0.91	0.83	0.87	0.90	0.86	0.92	0.92	0.89	0.93	0.87	0.84	0.90	0.89	0.83	0.82
15	0.86	0.93	0.88	0.92	0.90	0.90	0.82	0.87	0.89	0.85	0.91	0.91	0.88	0.92	0.86	0.83	0.89	0.88	0.83	0.81
16	0.86	0.92	0.89	0.92	0.91	0.90	0.81	0.86	0.88	0.84	0.90	0.90	0.88	0.91	0.86	0.83	0.89	0.88	0.81	0.81
17	0.87	0.90	0.87	0.90	0.88	0.89	0.87	0.88	0.89	0.85	0.88	0.92	0.86	0.92	0.86	0.84	0.87	0.88	0.88	0.86
18	0.87	0.90	0.87	0.90	0.88	0.89	0.87	0.88	0.89	0.85	0.88	0.92	0.86	0.92	0.86	0.84	0.87	0.88	0.88	0.86
19	0.86	0.91	0.89	0.92	0.89	0.90	0.83	0.86	0.85	0.85	0.88	0.90	0.87	0.91	0.86	0.83	0.88	0.87	0.84	0.84
20	0.87	0.95	0.92	0.96	0.94	0.93	0.81	0.88	0.89	0.85	0.92	0.92	0.89	0.92	0.87	0.83	0.90	0.89	0.81	0.80
21	0.87	0.90	0.87	0.90	0.88	0.89	0.87	0.88	0.89	0.85	0.88	0.93	0.86	0.92	0.86	0.84	0.87	0.88	0.88	0.86
22	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 3 shows an example hypothetical performance matrix for a single benchmark using this experimental methodology, where the 22 rows indicate distinct PGO binaries and the columns indicate the 20 distinct provided inputs. We use this table to describe the following terms that we employ to explain the PGO results in the rest of the paper.

PGO-Best refers to the best PGO performance across all PGO binaries and 20 inputs, and indicates the best gain from using PGOs for this program. This is the smallest ratio in the top 21 rows of Table 3. In contrast, *PGO-Worst* indicates the poorest overall PGO performance observed, and denotes adverse scenarios when the use of PGO results in low performance gain or even a loss (over O3). This is the highest ratio in the top 21 rows of Table 3.

PGO-Same describes the configuration where a binary customized by PGOs using a profile/input (say, A) is executed using the same input A. In such cases the profile used during PGOs is an ideal match to the input used during evaluation. *PGO-Same-Best* refers to the best performance obtained in the *PGO-Same* configuration. *PGO-Same-Worst* is the worst performance delivered by the *PGO-Same* configuration.

PGO-Cumul describes the configuration using the binary generated by the cumulative profile. Earlier studies show that a cumulative profile over several inputs may help PGOs deliver close to the ideal performance on multiple program inputs, while preventing adverse cases. *PGO-Cumul-Best* refers to the best performance achieved with the PGO-Cumul configuration, while *PGO-Cumul-Worst* describes the lowest performance in this configuration.

4 Experiments & Observations

In this section, we describe our analysis, explain our observations, and answer many interesting questions regarding the properties of PGOs in mainstream C compilers.

4.1 Performance Benefit from PGO

The goal of employing PGOs is to generate binaries customized to certain execution-time program profiles that can then deliver enhanced performance for those profiles. Since the same statically generated binary will be used for all program inputs, an important consideration is the performance impact of the customization process for program inputs that may not match the profile employed during PGO. In this work we measure the best-case and average-case performance benefit from PGOs in GCC and Clang. We study the performance impact of detrimental cases, when they happen. Compiler optimization and PGO techniques for C language compilers are extremely well-studied and published. Given that such studies are widely available to students and compiler writers, we determine if different mature mainstream C compilers show similar PGO-related performance traits.

Best-Case Performance Benefit from PGOs:
Figures 1 and 2 show the PGO performance results with the GCC and Clang compilers, respectively. The first bar for each benchmark in Figures 1 and 2 shows the *best* case performance impact from PGOs with the respective compilers. Thus, we see that PGOs can deliver significant performance gains for several programs, including benefits up to 32% for *bitcount*, 30% for *jpeg_c* and 21% for *pgp_d* with GCC, and benefits up to 16% for *pgp_d*, 15% for *jpeg_c*, and 13%

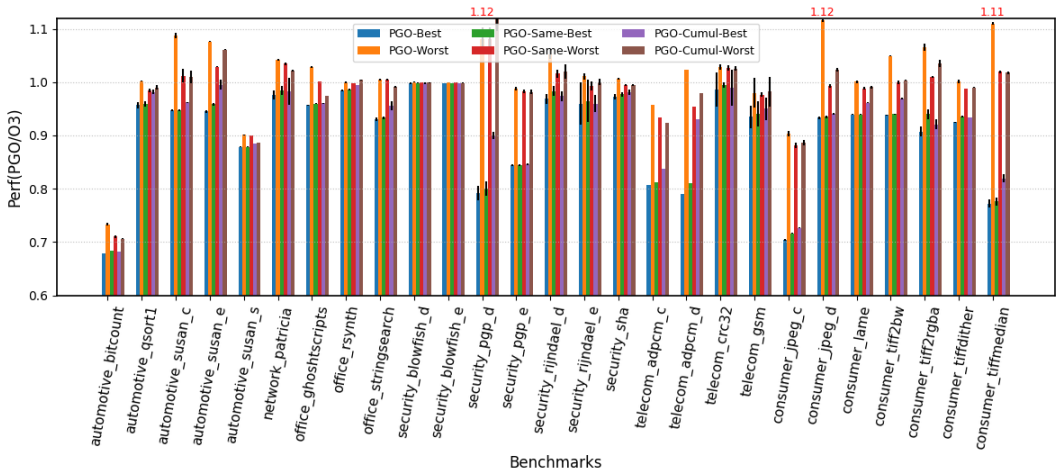


Figure 1. PGO-Best, PGO-Worst, PGO-Same, and PGO-Cumul results with GCC

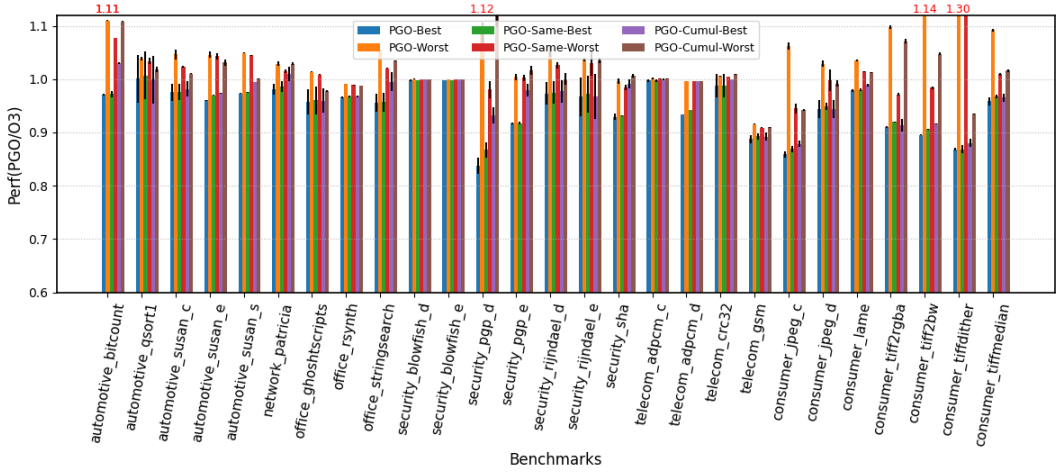


Figure 2. PGO-Best, PGO-Worst, PGO-Same, and PGO-Cumul results with Clang

for *tiffdither* with Clang. Also important, the PGO-Best performance was always at least as good as the baseline O3 (no-PGO) performance with both our compilers. We also find that not all programs benefit from PGOs, while the performance gain is meager for several others. This observation is consistent with prior research [34] and confirms the existing knowledge that PGO effectiveness is program and workload-dependent. The average PGO-Best performance gain over all our benchmarks is 8% with GCC, and 4% with Clang. Thus, this result gets us the first observation about PGOs in mainstream C/C++ compilers, which is that:

> **Observation #1**: As is generally believed, the application of PGOs can result in a significant performance gain for many programs.

Worst-Case Performance Benefit from PGOs:
The second bar in Figures 1 and 2 shows the PGO-Worst result, which is the maximum detrimental effect from PGOs when the program input during execution does not match the profile used during compilation. Unfortunately, we see that PGOs can have a large negative impact, delivering performance that is lower that the O3 performance in many cases, including losses of 8%, 12%, and 11% for *pgp_d*, *jpeg_d*, and *tiffmedian*, respectively with GCC, and degradations of up to 14%, 11%, and 8% for *tiff2bw*, *bitcount* and *pgp_d*, respectively with Clang. In several cases the performance of PGO-Worst is substantially lower than the default O3 optimization level. This result leads us to our second observation about PGOs in mainstream C/C++ compilers, which is that:

Observation #2: Application of PGOs with non-representative profiles can result in severe losses. Thus, realizing the gains, while limiting the adverse cases, remains an important concern with PGOs. This observation is also consistent with prior knowledge.

Ideal-Case Performance Benefit from PGOs:
The next two bars (bars 3 and 4) for each benchmark in Figures 1 and 2 show the performance range of the PGO-Same configuration for GCC and Clang, respectively. Remember that with PGO-Same the program is run with the same input as the profile used during binary customization. Since the performance benefit with PGOs is expected to heavily depend on the quality of the profile used, the PGO-Same configuration provides an ideal usage scenario with PGOs [27]. Correspondingly, we find that PGO-Same-Best achieves performance that is equal or close to the PGO-Best result in most of our cases with both compilers. However, counter-intuitively, we find that the PGO-Same configuration does not always produce the best results, and in the worst case it delivers performance that is equally or almost as poor as the overall PGO-Worst for each benchmark. This is a confusing result that we investigate further, and may require tuning the heuristics employed during compiler optimizations.

We further studied the PGO-Same results by comparing the number of times this configuration produces the best performance result for each corresponding benchmark. The results are shown in Table 4. We find that PGO-Same achieves the *best* performance in only about half the cases. Specifically, the PGO-Same configuration produces the best performance in 47% of the cases with Clang and 51% of the cases with GCC. For the cases identified as *Not Best*, the last column in Table 4 shows that PGO-Same performs poorer compared to PGO-Best by an average of 2.7% for Clang and 3.1% for GCC. These results indicate that the observed deltas reflect actual performance difference between PGO-Same and PGO-Best, rather than noise-induced variations. Interestingly, there is never a case when the performance provided by the PGO-Same scenario was not also produced by some other (seemingly, unrelated) program input/profile guiding the PGOs. This result with PGO-Same gives us to our third observation about PGOs in mainstream C/C++ compilers:

Observation #3: Even the ideal application of PGOs with a perfect profile may result in performance losses, a result that calls for further investigation by compiler developers.

Benefit from PGOs with Cumulative Profile:
Earlier research suggests that combining multiple program execution-time profiles into a single aggregate or *cumulative* profile allows PGOs to achieve good performance while avoiding adverse performance degradations in many cases [8]. Correspondingly, both GCC and Clang recommend mechanisms to generate cumulative profiles before deploying PGOs. For this work, we combine all our 20 unique profiles into the single cumulative profile for each benchmark.

Table 4. Analysis of PGO-Same

	PGO-Same is ...			
Compiler	Sole Best	Tied Best	Not Best	Delta(%)
LLVM	0	252	288	2.7
GCC	0	276	264	3.1

The last two bars for each benchmark in Figures 1 and 2 show the best and worst performance results with the PGO-Cumul configuration with the GCC and Clang compilers, respectively. We find that using cumulative profiles can mitigate to some extent, but not entirely, the worst losses caused by PGOs using non-representative profiles. Thus, while PGO-Cumul-Worst still results in a performance loss of up to 11% (for *pgp_d*) with GCC and 12% (again, for *pgp_d*) with Clang, the degradations are often lower than the worst-case results seen with PGO-Worst. Unfortunately, employing cumulative profiles also reduces the best-case performance gains from PGOs in most cases, presumably due to its *averaging* effect. That is, the PGO-Cumul-Best performance is typically lower than the PGO-Best performance with both the GCC and Clang compilers. This is an interesting result that challenges the rationale for generating and using the cumulative profile. These results with the cumulative profile gives us to our fourth observation about PGOs:

Observation #4: While the aggregation of profile information from multiple program runs can mitigate the worst-case losses from PGOs, they also have an adverse effect on the best performance from PGOs. Thus, the issue of generating the ideal profile data to use for PGOs remains an open research question.

Performance Corelation Between Compilers:
Issues regarding the types of program profile information to collect and strategies to apply them during optimizations to benefit performance for C/C++ programs have been very well studied over many years. Therefore, it may be reasonable to expect that different mainstream C/C++ compilers guided by such extensive past research would implement similar strategies and achieve comparable benefits (and perhaps, losses) for PGOs. However, after comparing Figures 1 and 2 we find that the PGO performance results for the GCC and Clang compilers bear little noticeable correlations with each other. Some benchmarks, like the two symmetric key encryption algorithms, *blowfish* and *rijndael*, do not benefit from PGOs with both compilers. Some other programs, like *pgp* and *jpeg_c*, show significant best-case improvement

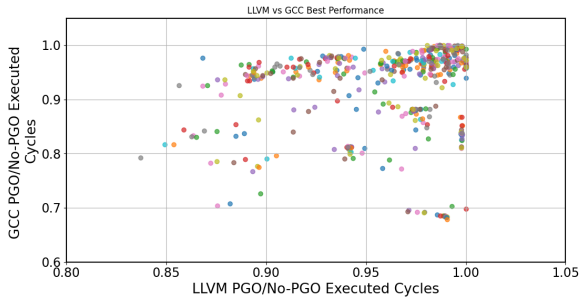


Figure 3. The level of benefit a program input receives from PGOs with one compiler is not a meaningful predictor of the level of benefit that will be received with another compiler

with PGOs for both compilers. Yet, the differences dominate the similarities. On average, GCC achieves higher best-case performance gains from PGOs. Several programs, like *bitcount* and *adpcm_c*, see significant performance gains with GCC, but not with Clang. In contrast, other programs, like *tiffdither*, gain more with Clang in the best-case.

To study these divergent results further, Figure 3 plots the best PGO performance (compared to O3) achieved by GCC on the Y-axis for each benchmark-input pair to the corresponding PGO performance with Clang on the X-axis. The cluster of points in the upper-right corner indicates that PGOs offer small to no benefits to many benchmark-inputs pairs with both GCC and Clang. Points in the lower right corner show cases that improve with GCC, but not with Clang. Points in the upper left corner show cases that improve with Clang, but not with GCC. Thus, this scatter plot again confirms the poor correlation between the GCC and Clang performance results with PGO.

> **Observation #5**: For GCC and Clang, the performance delivered by PGOs does not correlate well with each other. Such behavioral divergence, likely due to differences in implementation, applied heuristics and optimizations, may indicate opportunities to learn and improve for both compilers.

4.2 Code Generation with PGOs

In this work, we do not study the compiler source code to understand what and how the different optimization passes in each compiler employ profile data during their code transformation decisions. However, we study the behavior of individual optimization passes with (and without) PGOs, the sensitivity of the PGO passes to differences in supplied profile data, in terms of generated code variants, and differences between compilers. We present these results in this section.

Performance Sensitivity of Optimizations:
Compilers implement many optimizations passes. Several of these passes use profile data, when available and directed, to improve their effectiveness. We present observations from an experiment we devised to understand the sensitivity of individual optimization passes to profile data. More specifically, we attempt to study how individual optimization passes react to profile data, positively, negatively, or not at all.

GCC provides 154 command-line flags to control the application of individual optimization passes. Clang, however, does not similarly expose flags to enable/disable optimizations. The Clang frontend, invokes the *opt* tool to apply machine-independent transformations, followed by *llc* for machine-specific transformations. We update the Clang source code (including, *opt* and *llc*) to introduce 85 new command-line flags to *disable* individual optimization passes.[1]

Our experiment starts with the -O3 configuration, and uses the command-line flags to *successively disable* transformations, one by one, cumulatively, until the final configuration has all optimizations disabled (ideally, -O0). We then have two variants for each such configuration, one with PGOs ON, the other with PGOs OFF. Remember that with PGOs ON, each benchmark can be compiled with one of its 21 different profiles, and executed with any of 20 different inputs. Each benchmark-profile-input configuration (with PGOs ON/OFF) requires the generation of a distinct binary.

For better illustration, we plot the results of this experiment for only a small sample (8) of representative benchmark-input configurations [2]. For experiments with PGOs ON, we select some best-case input/profile configurations (indicated by the '(B)' suffix after the benchmark name in the figure legend), and other worst-case input/profile configurations (as indicated by the '(W)' benchmark suffix). The set of benchmarks plotted differs for GCC and Clang.

Figures 4(a) and 4(b) present the impact of cumulatively disabling individual optimizations on program performance, for GCC and Clang, respectively. Each data-point plots the ratio of O3/O0 execution-time cycles (on y-axis) when 'n' flags (on X-axis) are cumulatively disabled. With GCC, we find that even after disabling all 154 flags, performance does not reach -O0 , which suggests the presence of many *hidden* optimization passes in GCC that are not exposed to user control via command-line flags. While that is not the case with our updates to Clang, we instead find a few optimizations with a dominant impact on performance. Overall, we observe that with both compilers, (a) several optimizations contribute to performance movements, with many also causing performance losses, and (b) the set of optimizations causing performance gains or losses varies for each configuration, which provides the compelling basis for the considerable research in optimization *pass selection* [7, 22, 36].

[1]The list of 154 GCC and 85 Clang optimization command-line flags are listed in Appendix Table 5 and 6

[2]We ran many other benchmark-configurations, and the performance trends displayed here represent the overall trends we observed.

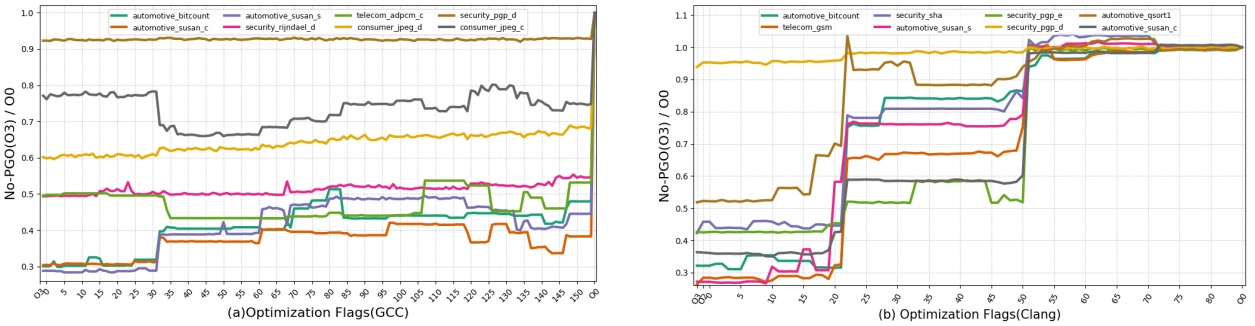


Figure 4. Performance impact of cumulatively disabling individual -O3-level optimization passes (no PGOs)

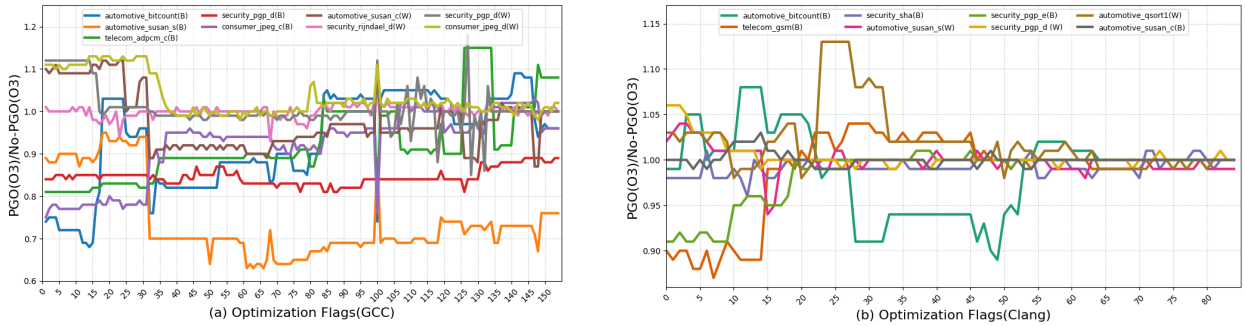


Figure 5. Performance impact of profile information on individual optimizations. At each data-point on X-axis, 'n' passes are disabled, and the PGO/no-PGO execution-time cycles ratio for the program is plotted on the Y-axis. The '(B)' and '(W)' suffix indicate the 'best' or 'worst' performing configuration, respectively, for that benchmark. There is one additional curve in the GCC graph since we plot both the 'best' and 'worst' performing configurations for 'pgp_d', as a special case.

Figures 5(a) and 5(b) show how individual optimization passes are affected by the presence of profile data, for GCC and Clang, respectively. Each data-point plots the impact of profile data on cumulative optimization performance. Data-point '0' shows the overall impact of profile data (PGO/no-PGO cycles) on program performance (all optimization passes are ON). Each successive point after that plots the same ratio with one additional pass OFF. Thus, a positive or negative impact of profile data on pass 'n' will show as a performance gain or loss, respectively, over pass 'n-1'; no impact will keep the point 'n' steady with data-point 'n-1'. Thus, these figures reveal some interesting results: (a) Many optimization passes employ profile data to improve their effectiveness, (b) There is no discernible trend in how profile data impacts a specific optimization pass.

There are several additional points to note. First, given the magnified scale on the Y-axis used in these plots, while minor shifts in the graph may be within the range of standard deviation (error bars not plotted for clarity), there are also many significant shifts in performance for each benchmark. Second, this is a substantial experiment that required extensive updates to the Clang compiler source code, and significant compute resources. Yet, the experiment is still exploratory and may need to be developed further to study additional optimization effects, including optimization interactions enabling or disabling opportunities for other optimization passes.

Overall, the impact of profile data on optimization behavior appears quite fickle, often characterized by significant performance swings during the intermediate stages of optimization. There is no trend of continuous performance gain or loss with PGOs, for most of our benchmarks. These results reveal our sixth observation about PGOs in C/C++ compilers:

> **Observation #6**: The effect of profile data on optimizations may be subject to (un)predictability issues similar to, and perhaps even more severe than, those afflicting general compiler optimization behavior, and may require solutions similar to *pass selection*, i.e., selectively determine which passes employ what profile data, if any, for each workload and input. This observation may also help explain the non-intuitive results seen earlier in Section 4.1, especially Observation #3.

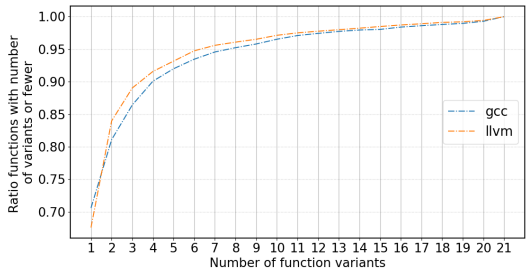


Figure 6. The majority of functions only have a small number of compiled variants

Number of Distinct Function Variants Generated:
PGOs use profile data to customize the generated code to maximize the execution-time program performance. In this section we attempt to understand the sensitivity of this customization. In other words, do (small) changes in the collected profile data bias the PGOs differently to generate different code customizations in each case?

For each benchmark in this work, we have gathered 21 different run-time profiles – one for each of the 20 individual inputs provided by MiDatasets, and one cumulative profile. We use these 21 profiles to generate 21 distinct binaries using PGOs. Thus, each program function can have 21 potential distinct variants. However, some of these generated binaries may be identical. We compare these 21 binary instances with each other to determine the actual number of *distinct* variants for each function in our benchmark set.

Directly comparing the binary or the corresponding assembly versions of different variants of a function may exhibit *false* differences due to inconsequential differences in the general-purpose registers or memory addresses and offsets assigned by the compiler. To prevent triggering such false variants, we first use the Ghidra disassembler to translate the binary code for each function variant to higher-level assembly code. Ghidra is an open-source reverse engineering framework, developed and open-sourced by the U.S. National Security Agency [35]. During this translation process, we normalize all memory addresses, register values, and non-IP-relative immediate values in the code by replacing them with constant patterns. Next, we use Ghidra's API to partition the function code into its constituent *basic blocks*.

To compare any two variants for a function, we compute the mapping between their basic blocks, and then compare each instruction in every corresponding block. We partition the 21 instances of each function into distinct *equivalence classes*, such that all instances in each class have identical block mappings and instructions, while either the block layout or assembly instructions or both are different across the different equivalence classes.

For several programs in our benchmark set, there are functions that are never reached during any execution. We remove these functions from consideration for this experiment as they do not generate any profile information. There are a total combined 5854 executed functions in our 27 benchmarks [3]. Of these 5854 executed functions, 1598 have a single basic block in the O3 program binary. We found that none of these single-block functions have more than one variant. Therefore, we do not plot these functions in any graphs in this section to more clearly see the trends for the remaining more interesting multi-block functions.

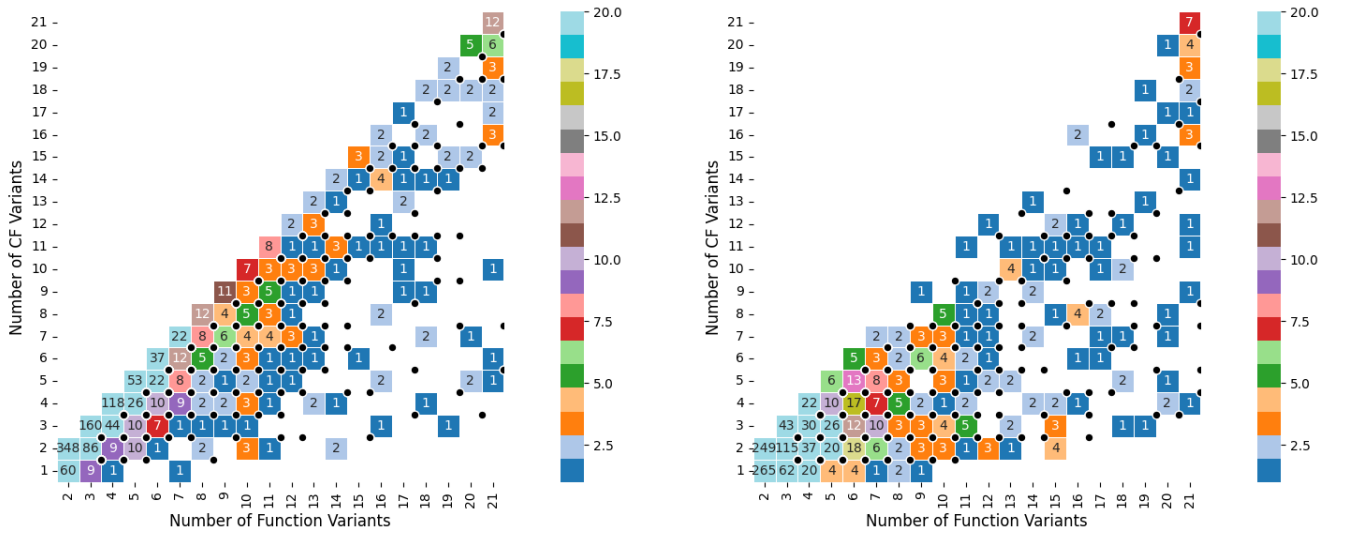
Figure 6 plots the ratio of the number of multi-block functions with N distinct variants to the total number of executed functions, in ascending values of N on the X-axis. Thus, we find that, even after removing the small single-block functions, most other functions still only generate a very small number of distinct variants. For 87.9% of the functions, LLVM generates 2 or fewer variants. With GCC, 86% of the functions have 3 or fewer variants. We also see that, in general, LLVM generates fewer variants compared to GCC.

Importantly, we also observe that, albeit small, there is a significant fraction of functions that have a very high number of variants, with several where the compiler generates different codes for every profile that we could provide. This abundance of generated variants for a small fraction of functions is also a consequential result, which while reinforcing the Pareto principle (the 80/20 or 90/10 rule) also demonstrates the vitality and depth of the customizations that compilers can perform in response to compelling variations in run-time behavior. These results studying the static code variants generated for each function by GCC and Clang reveal the seventh observation about PGOs:

> **Observation #7**: Compilers generate a small number of variants for most functions. Yet, PGOs are sensitive to changes in profile data, and can generate many distinct variants in several cases.

Dominant category of PGOs:
Compiler transformations can be categorized based on whether or not they update the program control-flow (CF). Many PGOs, like function inlining, loop unrolling, block layout, and hot/cold code splitting, often update the function/program control-flow layout, while a few, such as register allocation, typically do not. In order to better understand and compare the category of PGOs deployed by our two compilers, each plot in Figure 7 shows a heat map of the number of CF variants for each class of function variants. Thus, the leftmost cells in Figure 7(a) specify that of the (60+348=)408 functions that have 2 variants with PGOs for GCC, 60 functions have

[3]Due to compiler optimizations, such as *inlining* and *hot code splitting*, the number of executed function differ in GCC and Clang. Here we report the intersection of the function names provided by the two compilers.



(a) with GCC (b) with Clang

Figure 7. Heat map of number of function variants with different block-level control-flows for GCC (left) and Clang (right)

a single CF, while 348 have 2 distinct CFs, across the 2 variants. Likewise, for the 255 functions with 3 PGO variants in Figure 7, 9 of them reveal a single CF across the 3 variants, 86 have 2 different CFs, and 160 of them have a different CF across their 3 PGO variants. The X-axis for the plots in Figure 7 starts at 2 because we do not show single-block functions, as mentioned earlier.

We make several interesting observations from these graphs. First, they show a different clustering of the cells. While we see the cells clustering along the primary diagonal in Figure 7(a) with GCC, a cluster in the bottom-left corner is more conspicuous in Figure 7(b) with Clang. These graphs indicate that there are more functions with a smaller number of CFs with Clang. Likewise, we can also infer that control-flow altering PGO passes seem to be more active in GCC, compared to Clang. Interestingly, with both compilers, there were instances of functions with 20 or more distinct control-flow variants. However, the two compilers appear to differ in the compiler optimizations most influenced by profile data. These results studying the CF variants generated for each function by GCC and Clang reveal the eighth observation about PGOs in mainstream C/C++ compilers:

> **Observation #8**: A dissimilar set of optimization passes seem to be most affected by profile data in the two compilers, providing an opportunity for compilers to learn for each other.

Best performance on all inputs: Figure 8 plots the number of *statically* distinct program executables generated for each benchmark due to differences in profile data. Remember that, in our experiments, we have 21 possible binary variants for each benchmark and 20 (plus one cumulative) possible inputs. Since the data for this figure examines static variants at the whole-program level, even one static variation in one function is sufficient to differentiate it from all others program binaries. Thus, from this figure we see that, for most programs, each individual binary generated by PGOs using a different run-time profile results in different static code.

Given the considerable diversity in static code instances produced by PGOs in response to differing run-time profiles, we attempt to find the minimum number of program binaries necessary to achieve the best performance over all inputs to each benchmark. We use an approximate greedy algorithm for this experiment that first finds and orders each binary variant based on the number of program inputs for which it attains the best performance. Of course, there could be several binaries that achieve the best execution time for any given program input. Our algorithm first selects the binary that attains the best performance on the most number of inputs, followed greedily by the next binary that achieves the best performance on the most number of *remaining* inputs, and so on until all our 20 inputs are covered.

Figure 9 plots the minimum number of program binary variants our algorithm finds to achieve best performance on all inputs for each benchmark and for both compilers, GCC and Clang. Interestingly, we find that all our benchmarks can achieve the best performance across all inputs with five or fewer binary executables. This result suggests that static differences in code produced due to differing profiles may not always translate into different execution time performance. For several small benchmarks, such as *pgp_e*, *rijndael*

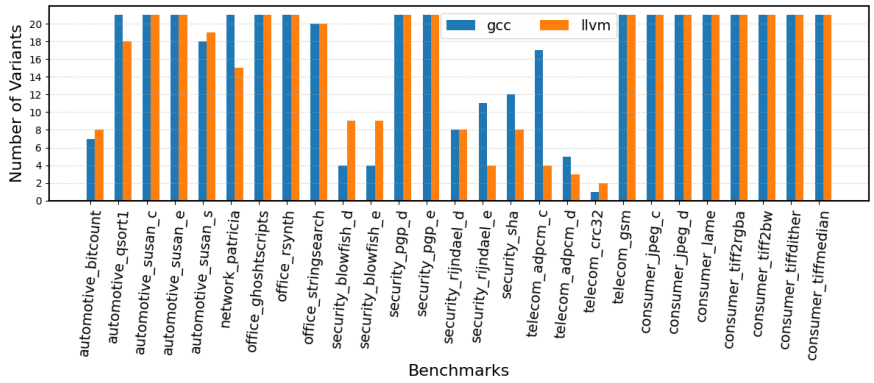


Figure 8. Number of static whole-program variants

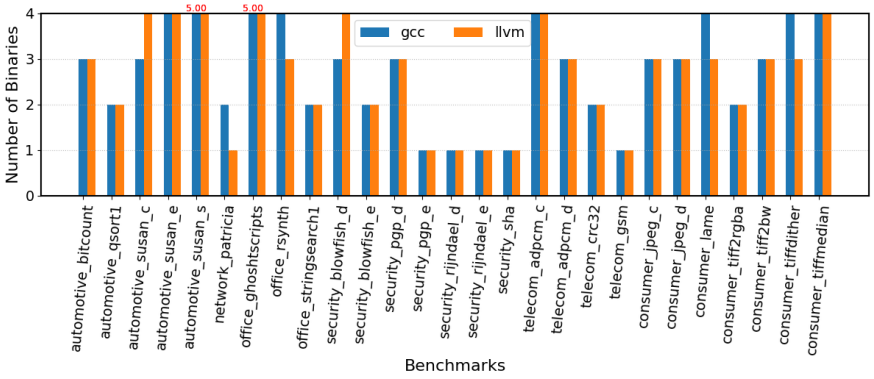


Figure 9. Smallest number of binaries to achieve best performance across all inputs

and *sha*, we found that a single binary can attain best performance for all inputs with both out compilers. While the two compilers continue to display variations, there is broad overall consistency in these results. Our analysis in this section reveal our final observation about PGOs in mainstream C/C++ compilers, which is that:

> **Observation #9**: Only a fraction of the static code customizations triggered by run-time profile data appear to prove consequential for program performance. The ability to identify and correlate them with patterns in the code input may make it feasible to generate a single multi-variant binary that can obtain best performance on all possible program inputs [14, 44].

5 Future Work

There are multiple avenues for future work. First, we plan to develop mechanisms to independently assess the diversity of the MiBench dataset we used for our research. Second, we will explore techniques similar to those employed in the *phase selection* research so optimizations use profile data more selectively only when it is likely to improve program performance. Third, earlier studies show the faults of profile data aggregation [30], which may partially explain our results with the cumulative profile. We plan to develop new approaches to overcome such challenges. Fourth, given the divergence in our results across the two compilers, we will study if transferring optimization strategies from one compiler to another is feasible and can lead to improvements.

6 Conclusions

In this work we comprehensively study the properties, behavior and benefits of PGOs in mainstream C/C++ compilers. As part of our investigation, we identify and reveal nine *observations* about PGOs in our framework, many of which were remarkable and unanticipated. Several of our observations, including the behavior of the *perfect* and the cumulative profiles, the divergence in PGO behavior across mature compilers, and the behavior of individual optimizations with profile data inform exciting avenues for future research. We hope that resolving these aspects can result in further, consistent, and explainable gains from PGOs across programs and compilers.

Table 5. Appendix A: GCC Optimization Flags

Index	GCC Flags	Index	GCC Flags
1	-fno-aggressive-loop-optimizations	78	-fno-prefetch-loop-arrays
2	-fno-align-functions	79	-fno-printf-return-value
3	-fno-align-jumps	80	-fno-ree
4	-fno-align-labels	81	-fno-reg-struct-return
5	-fno-align-loops	82	-fno-rename-registers
6	-fno-allocation-dce	83	-fno-reorder-blocks
7	-fno-asynchronous-unwind-tables	84	-fno-reorder-blocks-and-partition
8	-fno-auto-inc-dec	85	-fno-reorder-functions
9	-fno-bit-tests	86	-fno-rerun-cse-after-loop
10	-fno-branch-count-reg	87	-fno-sched-critical-path-heuristic
11	-fno-caller-saves	88	-fno-sched-dep-count-heuristic
12	-fno-code-hoisting	89	-fno-sched-group-heuristic
13	-fno-combine-stack-adjustments	90	-fno-sched-interblock
14	-fno-compare-elim	91	-fno-sched-last-insn-heuristic
15	-fno-cprop-registers	92	-fno-sched-rank-heuristic
16	-fno-crossjumping	93	-fno-sched-spec
17	-fno-cse-follow-jumps	94	-fno-sched-spec-insn-heuristic
18	-fno-dce	95	-fno-sched-stalled-insns-dep
19	-fno-defer-pop	96	-fno-schedule-fusion
20	-fno-delete-null-pointer-checks	97	-fno-schedule-insns2
21	-fno-devirtualize	98	-fno-short-enums
22	-fno-devirtualize-speculatively	99	-fno-shrink-wrap
23	-fno-dse	100	-fno-shrink-wrap-separate
24	-fno-early-inlining	101	-fno-signed-zeros
25	-fno-expensive-optimizations	102	-fno-split-ivs-in-unroller
26	-fno-forward-propagate	103	-fno-split-loops
27	-fno-fp-int-builtin-inexact	104	-fno-split-paths
28	-fno-function-cse	105	-fno-split-wide-types
29	-fno-gcse	106	-fno-ssa-backprop
30	-fno-gcse-after-reload	107	-fno-ssa-phiopt
31	-fno-gcse-lm	108	-fno-stack-clash-protection
32	-fno-guess-branch-probability	109	-fno-stdarg-opt
33	-fno-hoist-adjacent-loads	110	-fno-store-merging
34	-fno-if-conversion	111	-fno-strict-aliasing
35	-fno-if-conversion2	112	-fno-strict-volatile-bitfields
36	-fno-indirect-inlining	113	-fno-thread-jumps
37	-fno-inline	114	-fno-toplevel-reorder
38	-fno-inline-atomics	115	-fno-trapping-math
39	-fno-inline-functions	116	-fno-tree-bit-ccp
40	-fno-inline-functions-called-once	117	-fno-tree-builtin-call-dce
41	-fno-inline-small-functions	118	-fno-tree-ccp
42	-fno-ipa-bit-cp	119	-fno-tree-ch
43	-fno-ipa-cp	120	-fno-tree-coalesce-vars
44	-fno-ipa-cp-clone	121	-fno-tree-copy-prop
45	-fno-ipa-icf	122	-fno-tree-cselim
46	-fno-ipa-icf-functions	123	-fno-tree-dce
47	-fno-ipa-icf-variables	124	-fno-tree-dominator-opts
48	-fno-ipa-modref	125	-fno-tree-dse
49	-fno-ipa-profile	126	-fno-tree-forwprop
50	-fno-ipa-pure-const	127	-fno-tree-fre
51	-fno-ipa-ra	128	-fno-tree-loop-distribute-patterns
52	-fno-ipa-reference	129	-fno-tree-loop-distribution
53	-fno-ipa-reference-addressable	130	-fno-tree-loop-if-convert
54	-fno-ipa-sra	131	-fno-tree-loop-im
55	-fno-ipa-stack-alignment	132	-fno-tree-loop-ivcanon
56	-fno-ipa-vrp	133	-fno-tree-loop-optimize
57	-fno-ira-hoist-pressure	134	-fno-tree-loop-vectorize
58	-fno-ira-share-save-slots	135	-fno-tree-partial-pre
59	-fno-ira-share-spill-slots	136	-fno-tree-phiprop
60	-fno-isolate-erroneous-paths-dereference	137	-fno-tree-pre
61	-fno-ivopts	138	-fno-tree-pta
62	-fno-jump-tables	139	-fno-tree-reassoc
63	-fno-lifetime-dse	140	-fno-tree-scev-cprop
64	-fno-loop-interchange	141	-fno-tree-sink
65	-fno-loop-unroll-and-jam	142	-fno-tree-slp-vectorize
66	-fno-lra-remat	143	-fno-tree-slsr
67	-fno-math-errno	144	-fno-tree-sra
68	-fno-move-loop-invariants	145	-fno-tree-switch-conversion
69	-fno-omit-frame-pointer	146	-fno-tree-tail-merge
70	-fno-optimize-sibling-calls	147	-fno-tree-ter
71	-fno-optimize-strlen	148	-fno-tree-vrp
72	-fno-partial-inlining	149	-fno-unroll-loops
73	-fno-peel-loops	150	-fno-unswitch-loops
74	-fno-peephole	151	-fno-var-tracking
75	-fno-peephole2	152	-fno-var-tracking-assignments
76	-fno-plt	153	-fno-version-loops-for-strides
77	-fno-predictive-commoning	154	-fno-web

Table 6. Appendix B: Clang Optimization Passes

Index	Clang Pass Names	Index	Clang Pass Names
1	-sroa	44	-constmerge
2	-ipsccp	45	-lcssa
3	-globalopt	46	-stack-coloring
4	-function-attrs	47	-peephole-opt
5	-bdce	48	-register-coalescer
6	-tailcallelim	49	-machine-scheduler
7	-indvars	50	-machine-cp
8	-sccp	51	-greedy
9	-early-cse	52	-fast-isel
10	-licm	53	-branch-folder
11	-loop-rotate	54	-block-placement
12	-loop-idiom	55	-cfi-instr-inserter
13	-div-rem-pairs	56	-machine-licm
14	-always-Inline	57	-unreachable-mbb-elimination
15	-simplifycfg	58	-break-false-deps
16	-loop-simplify	59	-x86-cmov-conversion
17	-instcombine	60	-stack-slot-coloring
18	-globaldce	61	-x86-fixup-LEAs
19	-reassociate	62	-x86-fixup-bw-insts
20	-mem2reg	63	-machine-cse
21	-correlated-propagation	64	-machine-dce
22	-gvn	65	-machine-sink
23	-dse	66	-post-ra-pseudos
24	-instsimplify	67	-early-tailduplication
25	-inferattrs	68	-early-ifcvt
26	-loop-unroll	69	-postra-machine-licm
27	-aggresive-instcombine	70	-postra-machine-sink
28	-jump-threading	71	-loop-reduce
29	-loop-vectorize	72	-consthoist
30	-adce	73	-cgp
31	-elim-avail-extern	74	-partially-inline-libcalls
32	-memcpyopt	75	-implicit-null-checks
33	-inliner-wrapper	76	-mergeicmps
34	-deadargelim	77	-cfi-fixup
35	-loop-instsimplify	78	-select-optimize
36	-expand-large-div-rem	79	-x86-fixup-setcc
37	-infer-alignment	80	-machine-combiner
38	-called-value-propagation	81	-processimpdefs
39	-slp-vectorizer	82	-shrink-wrap
40	-loop-load-elim	83	-lrshrink
41	-mldst-motion	84	-lower-global-dtors
42	-loop-deletion	85	-replace-with-veclib
43	-vector-combine		

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 85–96. doi:10.1145/258915.258924

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. 1997. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 357–390. doi:10.1145/265924.265925

[3] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (2005), 449–466. doi:10.1109/JPROC.2004.840305

[4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, Minnesota, USA) *(OOPSLA '00)*. Association for Computing Machinery, New York, NY, USA, 47–65. doi:10.1145/353171.353175

[5] Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. 2000. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. Association for Computing Machinery, New York, NY, USA, 52–64. doi:10.1145/351397.351416

[6] Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 168–179. doi:10.1145/378795.378832

[7] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *ACM Comput. Surv.* 51, 5, Article 96 (Sept. 2018), 42 pages. doi:10.1145/3197978

[8] Paul Berube and Jose Nelson Amaral. 2012. Combined profiling: A methodology to capture varied program behavior across multiple inputs. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12)*. IEEE Computer Society, USA, 210–220. doi:10.1109/ISPASS.2012.6189227

[9] P. P. Chang and W.-W. Hwu. 1989. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89)*. Association for Computing Machinery, New York, NY, USA, 246–257. doi:10.1145/73141.74840

[10] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. 1992. Profile-guided automatic inline expansion for C programs. *Softw. Pract. Exper.* 22, 5 (May 1992), 349–369. doi:10.5555/138720.138721

[11] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. 1991. Using profile information to assist classic code optimizations. *Software: Practice and Experience* 21, 12 (1991), 1301–1321. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211204 doi:10.1002/spe.4380211204

[12] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. 1996. Profile-driven instruction level parallel scheduling with application to super blocks. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. IEEE Computer Society, USA, 58–67. doi:10.5555/243846.243858

[13] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/2854038.2854044

[14] Peng-fei Chuang, Howard Chen, G Hoflehner, D Lavery, and W Hsu. 2007. Dynamic profile driven code version selection. In *Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*. Citeseer.

[15] Michał Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. 2000. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 13–26. doi:10.1145/349299.349306

[16] Clang/LLVM. Accessed 2025. Clang Compiler User's Manual. https://clang.llvm.org/docs/UsersManual.html

[17] Thomas M. Conte, Kishore N. Menezes, and Mary Ann Hirsch. 1996. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. IEEE Computer Society, USA, 36–45. doi:10.1109/MICRO.1996.566448

[18] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. 1997. ProfileMe: hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (Research Triangle Park, North Carolina, USA) *(MICRO 30)*. IEEE Computer Society, USA, 292–302. doi:10.1109/MICRO.1997.645821

[19] Joseph A. Fisher and Stefan M. Freudenberger. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 85–95. doi:10.1145/143365.143493

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

[21] Grigori Fursin, John Cavazos, Michael O'Boyle, and Olivier Temam. 2007. MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization. In *Proceedings of the 2Nd International Conference on High Performance Embedded Architectures and Compilers* (Ghent, Belgium) *(HiPEAC'07)*. Springer-Verlag, Berlin, Heidelberg, 245–260. doi:10.5555/1762146.1762170

[22] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, Michael O'Boyle, Phil Barnard, Elton Ashton, Eric Courtois, and François Bodin. 2008. MILEPOST GCC: machine learning based research compiler. *Proceedings of the GCC Developers' Summit 2008* (06 2008).

[23] GCC. Accessed 2025. GCC online documentation. https://gcc.gnu.org/onlinedocs/

[24] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. doi:10.1145/800230.806987

[25] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 3–14. doi:10.1109/WWC.2001.15

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

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

[28] Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. 2024. Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 322–333. doi:10.1109/CGO57630.2024.10444807

[29] Wenlei He, Hongtao Yu, Lei Wang, and Taewook Oh. 2024. Revamping Sampling-Based PGO with Context-Sensitivity and Pseudo-instrumentation. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization* (Edinburgh, United Kingdom) *(CGO '24)*. IEEE Press, 322–333. doi:10.1109/CGO57630.2024.10444807

[30] Michael R. Jantz, Forrest J. Robinson, and Prasad A. Kulkarni. 2016. Impact of Intrinsic Profiling Limitations on Effectiveness of Adaptive Optimizations. *ACM Trans. Archit. Code Optim.* 13, 4, Article 44 (Dec. 2016), 26 pages. doi:10.1145/3008661

[31] Erik Johansson and Sven-Olof Nyström. 2000. Profile-guided optimization across process boundaries. *ACM SIGPLAN Notices* 35, 7 (2000), 23–31.

[32] Donald E. Knuth. 1971. An empirical study of FORTRAN programs. *Software: Practice and Experience* 1, 2 (1971), 105–133. doi:10.1002/spe.4380010203

[33] Scott Mahlke, Tipp Moseley, Richard Hank, Derek Bruening, and Hyoun Kyu Cho. 2013. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, USA, 1–10. doi:10.1109/CGO.2013.6494982

[34] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. 2021. VESPA: static profiling for binary optimization. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 144 (Oct. 2021), 28 pages. doi:10.1145/3485521

[35] National Security Agency. [n. d.]. Ghidra. GitHub repository. https://github.com/NationalSecurityAgency/ghidra.

[36] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. 2016. A graph-based iterative compiler pass selection and phase ordering approach. *SIGPLAN Not.* 51, 5 (June 2016), 21–30. doi:10.1145/2980930.2907959

[37] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 16–27. doi:10.1145/93542.93550

[38] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. *SIGPLAN Not.* 25, 6 (June 1990), 16–27. doi:10.1145/93548.93550

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

[40] Alan Dain Samples. 1992. *Profile-driven compilation*. Ph. D. Dissertation. USA. doi:10.5555/145733 UMI Order No. GAX92-03707.

[41] Vatsa Santhanam and Daryl Odnert. 1990. Register allocation across procedure and module boundaries. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 28–39. doi:10.1145/93542.93551

[42] Serap Savari and Cliff Young. 1999. Comparing and combining profiles. In *Second Workshop on Feedback-Directed Optimization (FDO)*, Vol. 240. Citeseer.

[43] Michael D. Smith. 2000. Overcoming the challenges to feedback-directed optimization (Keynote Talk). In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO '00)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/351397.351408

[44] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. 2010. An Input-centric Paradigm for Program Dynamic Optimizations. *SIGPLAN Not.* 45, 10 (Oct. 2010), 125–139. doi:10.1145/1932682.1869471

[45] Maja Vukasovic and Aleksandar Prokopec. 2023. Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 20 (Dec. 2023), 64 pages. doi:10.1145/3612937

[46] David W. Wall. 1986. Global register allocation at link time. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, USA) *(SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 264–275. doi:10.1145/12276.13338

[47] David W. Wall. 1991. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 59–70. doi:10.1145/113445.113451

[48] John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. Association for Computing Machinery, New York, NY, USA, 78–87. doi:10.1145/337449.337483

[49] Youfeng Wu and James R. Larus. 1994. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture* (San Jose, California, USA) *(MICRO 27)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/192724.192725

[50] Cliff Young and Michael D. Smith. 1998. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (Dallas, Texas, USA) *(MICRO 31)*. IEEE Computer Society Press, Washington, DC, USA, 115–123. doi:10.5555/290940.290968

[51] Alexander Zaitsev. 2025. Github: Awesome-PGO. https://github.com/zamazan4ik/awesome-pgo