Towards Threading the Needle of Debuggable Optimized Binaries

Cristian Assaiante*, Simone Di Biasio*, Snehasish Kumar†,
Giuseppe Antonio Di Luna*, Daniele Cono D'Elia*, and Leonardo Querzoni*

*Sapienza University of Rome, Rome, Italy
†Google, Mountain View, USA
{assaiante, delia, diluna, querzoni}@diag.uniroma1.it, sneaky@google.com

***Abstract*—Compiler optimizations may lead to loss of debug information, hampering developer productivity and techniques that rely on binary-to-source mappings, such as sampling-based feedback-directed optimization. While recent endeavors exposed debug information correctness and completeness bugs in compiler transformations, understanding where a complex optimizing pipeline "loses" debug information is an understudied problem.**

In this paper, we first rectify accuracy issues in methods for measuring the availability of debug information, and show that the synthetic programs evaluated so far lead to metric values that differ from those we observe for real-world programs. Building on this, we present DEBUGTUNER, a framework for systematically analyzing the impact of individual compiler optimization passes on debug information, and assemble a test suite of programs for collecting more realistic metrics. Using DEBUGTUNER and the test suite, we identify transformations in gcc and clang that cause more debug information loss, and construct modified optimization levels that improve debuggability while retaining competitive performance. We obtain levels that outperform gcc's `Og` for both debuggability and performance, and make recommendations for constructing an `Og` level for clang. Finally, we present a case study on AutoFDO where, by disabling selected passes in the profiling stage, the final optimized binary is more performant due to the improved quality of the binary-to-source mapping.

***Index Terms*—Debugging, Optimizing compilers, Profiling**

I. INTRODUCTION

In recent years, several endeavors [1]–[4] have revived interest on the long-standing issue of symbolically debugging optimized code [5]–[8]. These efforts focus on uncovering bugs in optimizing compiler toolchains that lead to incomplete or incorrect debug information, degrading in turn the source-level debugging experience. This can mislead developers during debugging, reducing productivity and challenging crash diagnostics in production environments or whenever bugs can surface only with optimizations enabled [9], [10]. Similarly, inaccurate mappings between binary instructions and source locations hamper sample-based feedback driven optimization, leading to sub-optimal performance and energy wastage.

The accuracy and completeness of debug information depend on how well a compiler can preserve source-level context while applying optimizations. However, modern optimizations—critical for improving performance and reducing costs—can significantly transform the original program structure [11], complicating the task of maintaining mappings.

To mitigate this, compilers like gcc provide debug-friendly `Og` optimization pipelines that retain more debug information. A typical use case for them in industry is when developing new features, as debuggability aids developers (e.g., when tests fail) before doing highly optimized release builds. Compiler developers typically derive these pipelines from `O1` through careful tuning of optimizations—limiting aggressive transformations or disabling them entirely [12]–[15]. However, their design often relies on heuristic and expert knowledge, while a systematic method to evaluate debug information quality may enable more informed and effective tuning. The performance of `Og` binaries may also be insufficient, leaving behind users who must work with more performant binaries.

The setbacks of insufficient debug information extend beyond interactive debugging. Source-to-binary mappings are used by several compilation and security technologies. Most notably, AutoFDO [16], a system for feedback-directed optimization using hardware performance counters, depends on accurate mappings to associate runtime behavior with source code. Missing or imprecise mappings can cause such a system to miss optimization opportunities, reducing performance. Similarly, security tools [17]–[19] use debug information to enhance crash dump analysis and reconstruct control-flow graphs or vulnerable source locations from binary analysis; poor debug coverage limits their effectiveness. Hence, the ability to assess and improve debug symbol quality has broader implications for performance and security tooling.

In this paper, we first rectify accuracy issues in existing methods for measuring debug information availability—which hereafter we may interchangeably refer to also as *quality*.

Static methods [20] tend to overestimate it by including source elements a debugger cannot represent; dynamic ones [4] can underestimate it due to limitations in debug symbols formats. Our approach integrates dynamic trace collection with source-level analysis to extract precise variable definition ranges, making the debuggability measurements more sound.

Next, we show that real-world programs result in different metric values than the synthetic programs used in prior studies (e.g., [4]), and will focus on the former in the paper.

We then introduce DEBUGTUNER, a framework for measuring debug information quality within an optimizing compiler. DEBUGTUNER evaluates the effects of individual optimizations

(or *passes* for short) by selectively disabling them from standard optimization levels. Based on these measurements, we extract insights to construct compiler configurations that improve debuggability and we assess their performance.

One challenge in dynamic analysis of real-world programs is ensuring sufficient code coverage. Since the number of source lines stepped during a debug session depends on input diversity, having adequate test inputs is key to result significance. We identify in coverage-guided fuzzing one avenue to construct a test set of programs and inputs. We pick a heterogeneous set of programs tested as part of the OSS-Fuzz continuous fuzzing initiative [21] and rely on the high-coverage inputs that OSS-Fuzz built for them over time to derive our test inputs.

This pipeline lets us analyze large amounts of real-world code with minimal manual effort. Using this setup, we study gcc and clang to identify passes that cause debug information loss. We then leverage these insights to identify compiler configurations—derived from the canonical `Ox` ones—that improve debuggability for a reasonable performance penalty.

These insights can provide practical guidance for two key compiler design aspects. The first is building debug-friendly optimization pipelines. The benefits are readily apparent for clang, where such efforts are still in their infancy. Then even more with gcc's `Og`, which has been manually tuned for years, the configurations we derive from `O1` outperform it for both debuggability and performance. The second is improving passes: DEBUGTUNER helps compiler architects spot passes that cause more debug information loss and prioritize them for enhancements, quantifiable by repeating the analysis method.

Finally, we present a case study on how incomplete debug information limits AutoFDO. By disabling selected passes in the profiling collection stage, we are able to improve the performance of the profile-optimized binary, as AutoFDO can now better map the collected binary-level profiles to source statements, enabling subsequent better optimization decisions. This showcases the essence of our endeavor, as we preserved key debug information that benefits downstream users.

Contributions: The main contributions of this paper are:

- A framework to assess the impact of individual optimizations on debug symbol coverage for compiler tuning.
- Two methods: one to accurately measure debug information, and one to build a test suite of real-world programs.
- An empirical study on gcc and clang that identifies the most harmful optimizations and demonstrates the construction of debug-friendly pipelines.
- A case study on AutoFDO's sensitivity to debug information availability and its end-to-end performance impact.

We make available the code and evaluation materials of DEBUGTUNER at https://github.com/Sap4Sec/debugtuner.

Paper structure: Section II introduces our methodology for evaluating debug information availability, explaining the limitations of existing approaches. Section III presents the DEBUGTUNER framework. Section IV describes how to analyze real-world applications using fuzzing-generated inputs. Section V presents and discusses the experimental results from applying our framework to gcc and clang, including the AutoFDO case study. The paper ends with a discussion of limitations, future research directions, and related works.

II. MEASURING DEBUG INFORMATION QUALITY

Recent efforts [4], [20] have aimed at formalizing practical metrics for evaluating the availability (or *completeness*) of compiler-generated debug information in optimized binaries.

The method proposed by Assaiante et al. [4] performs dynamic analysis during debug sessions to identify source-level constructs, tracking two quantities for debuggability:

- *Availability of variables*, as the average ratio of variables visible with a value during debugging of the optimized program compared to the unoptimized baseline.
- *Line coverage*, computing the fraction of source lines that can be stepped through in both builds.

The *product of the two metrics*, introduced as a composite quality indicator, expresses the extent to which variables are accessible on the lines that can be stepped during debugging. Ranges for metric values are $[0, 1]$, with 0 indicating full loss.

In contrast, the work of Stinnett and Kell [20] explores a static approach. Rather than using the unoptimized binary as a baseline, it statically computes the definition ranges of variables from the source code and compares them to their debug symbol coverage in the binary. This approach addresses a specific limitation of DWARF [22] debug information noted in their study, where variables appear visible outside their live ranges, thereby skewing dynamic measurements like [4].

However, while the dynamic analysis is prone to *underestimating* debug information quality due to an inflated baseline from DWARF's limitations, static analysis can *overestimate* quality by counting also variables whose locations are defined in debug symbols, but that do not materialize during a debug session. Identifying such inaccuracies would require deeper binary-level reasoning, which static methods lack.

A hybrid method: To address these issues, we introduce a hybrid method that leverages both runtime behavior and source-level information. Like Assaiante et al., we use dynamic analysis to preserve an end-user perspective by relying on actual debugger traces. However, we refine the measurement of variable availability by narrowing each variable's expected definition range using static source analysis. This addresses the DWARF-related issue highlighted in [20], eliminating the source of underestimation in the dynamic metrics.

We tested this approach on the 5000 synthetic programs generated via Csmith [23] that both works used. We compiled them using gcc 12.2.0 and clang 20.1.0 at optimization levels `Og` (only for gcc, clang does not have an equivalent), `O1`, `O2`, and `O3`. For each binary, we computed the three core metrics for Assaiante et al.'s dynamic method, Stinnett et al.'s static method, and our hybrid method. We used gdb 13.2 and lldb 20.1.0 for acquiring debug traces.

Table I summarizes the results, computed as the geometric mean of the scores from each test program. Since the static method includes dead or unreachable code in its baseline, we also compute a refined variant, denoted as *static-dbg*, which filters out source lines not stepped in the unoptimized binary.

TABLE I
COMPARISON OF DIFFERENT METHODS TO COLLECT DEBUG INFORMATION METRICS ON REFERENCE SYNTHETIC PROGRAMS.

compiler	opt level	availability of variables				line coverage			product of metrics			
		static	static-dbg	dynamic	hybrid	static	static-dbg	dynamic, hybrid	static	static-dbg	dynamic	hybrid
gcc	Og	0.9220	0.9039	0.8563	0.9025	0.3587	0.5941	0.5533	0.3307	0.5370	0.4738	0.4993
	O1	0.9085	0.8891	0.8285	0.8814	0.2204	0.3646	0.3382	0.2002	0.3242	0.2802	0.2981
	O2	0.8316	0.8004	0.6549	0.6915	0.2075	0.2911	0.2680	0.1726	0.2330	0.1755	0.1853
	O3	0.8764	0.8523	0.6826	0.7152	0.1947	0.2741	0.2521	0.1707	0.2336	0.1721	0.1803
clang	O1	0.8683	0.8687	0.7931	0.8584	0.2370	0.3411	0.3128	0.2057	0.2963	0.2481	0.2685
	O2	0.9127	0.9127	0.8258	0.9020	0.2117	0.3033	0.2773	0.1932	0.2768	0.2290	0.2501
	O3	0.9159	0.9160	0.8282	0.9052	0.2106	0.3016	0.2757	0.1929	0.2763	0.2283	0.2495

This ensures the evaluation focuses only on code that would be observed during a real debugging session, eliminating irrelevant lines that a developer would never step through.

Refining the baseline has opposing effects: it lowers variable availability scores, as the removed lines generally contributed positively to the average, and it increases line coverage, since the baseline now excludes code that was never executed, resulting in a smaller denominator. For all subsequent comparisons between our method and the static approach, we use the static-dbg variant for a fair evaluation on the same, relevant code.

The availability of variables[1] is the metric most affected by static overestimation. With higher optimization levels, our results diverge further from static-dbg on gcc, increasing from a gap of 0.0014 at `Og` to 0.1089 at `O3`, indicating a growth of static overestimation. This highlights how much debug information shows as in the binary but is unusable (likely ill-formed or due to debugger bugs). On clang, in contrast, the differences remain small across all levels, with an average gap of $\sim$0.01. Compared to the dynamic method, our use of source-based corrections to the true visibility ranges to mitigate DWARF limitations consistently yields higher availability scores.

Line coverage is, by construction, unaffected by our source-based refinements. Thus, our approach and the dynamic one result into identical scores. The static-dbg variant, however, reports higher line coverage than the dynamic approaches. This is because its baseline is narrower, due to the static analysis missing some lines that are actually visited at runtime.

When analyzing the combined metric (i.e., the product of availability and line coverage), we observe that our scores fall between the static and dynamic data points. This confirms our method's ability to correct for the static overestimation and the dynamic underestimation, producing more balanced and realistic representation of debug information quality—which reflects what the end user would experience when debugging.

Finally, since the availability of variables is inherently dependent on line coverage (i.e., availability is measured only on covered lines), we adopt the product of the two as our main quality score for the paper, as it provides a robust and meaningful estimate of the overall debuggability of a binary.

Representative test cases for debug symbols: While the 5000 synthetic test cases offer full code coverage and controlled evaluation, they differ from real-world software. For example, Csmith-generated code often contains complex expressions and artificial constructs designed to stress compilers, which may end up entirely optimized away in compilation. In contrast, real-world applications exhibit more conventional control flow and variable use, making them a better target for studying debug information for how users will perceive it.

[1]This metric was designed to compare compiler versions for a given level, and not different levels (for example, in clang it increases at `O3` because there are fewer stepped lines). Levels can be compared using the product of metrics.

TABLE II
DEBUG INFORMATION QUALITY METRICS ON `libpng`.

compiler	opt level	avail. of vars	line coverage	product of metrics
gcc	Og	0.8417	0.8646	0.7277
	O1	0.8002	0.8120	0.6497
	O2	0.7387	0.6992	0.5165
	O3	0.6924	0.6772	0.4689
clang	O1	0.8819	0.7290	0.6429
	O2	0.8495	0.7111	0.6041
	O3	0.8483	0.7043	0.5974

Table II reports debug information metrics collected with our method for libpng, a representative subject for our evaluation from Section V-A. Compared to the synthetic benchmark, variable availability is generally lower. This is due to real-world functions typically using fewer variables, so each missing variable weighs more heavily on the average. The metric remain consistently lower on libpng with two exceptions: at `O2` on gcc (0.6915 vs. 0.7387), and at `O1` on clang (0.8584 vs. 0.8819). Moreover, it declines monotonically with increasing optimization levels, unlike synthetic programs, where availability at `O3` often exceeds that at `O2`.

Conversely, synthetic programs show significantly lower line coverage compared to libpng, averaging 0.3382 vs. 0.8120 at `O1` and 0.2521 vs. 0.4689 at `O3` on gcc. This suggests that much of the synthetic code is aggressively optimized away, unlike when processing real-world code. Furthermore, real-world code modules (11 in libpng) are often much larger than in synthetic programs, for which we measured on average 56.42 stepped lines at `O0`. Compilers do a better job than expected at preserving line stepping information, even under higher optimization. Due to this imbalance, also the resulting product metric tends to be higher in real-world code.

These observations reinforce that synthetic programs, while useful for stress-testing, may not faithfully represent the challenges of preserving debug information, and do not reflect the behavior observed in realistic software development scenarios.

For the 5000 synthetic programs, we computed the geometric standard deviation for all metrics to estimate per-program differences, obtaining values in the range $[1.08, 1.12]$ that indicate low variability and no overlap with the libpng data.

III. The DebugTuner Framework

This section introduces DebugTuner, a framework designed to evaluate the impact of individual compiler optimizations on the quality of debug information and to guide the construction of optimization pipelines that result in improved debuggability of the compiled code. As shown in Figure 1, DebugTuner comprises two main components: the first analyzes the effect of compiler optimization passes on debug information availability, while the second ranks and selects passes to build improved compiler configurations.

A. Debug Information Evaluation

This component of the framework evaluates the quality of debug information for a given program, using the hybrid approach defined in Section II, and analyzing the effect of disabling single optimization passes.

We assume to have access to a set of inputs that ensure good code coverage for the target benchmark. Later, in Section IV, we introduce our debug information test suite, made of 13 applications. A test suite is necessary to capture the diverse ways in which optimizations interact with source-level constructs across different code bases, enabling a more robust and generalizable evaluation of debug information quality.

While this component is applied to the entire suite, the results are obtained independently for each program in it. For simplicity, we describe the workflow for a single program. Given a program p and a set of inputs I_p for it, the analysis proceeds through four main stages:

1) *Build*: The program is compiled with `O0`, producing $unopt_p$; then, with the selected optimization levels (`O1`, `O2`, and `O3`, as well as `Og` when available), and with derived configurations where one individual pass from the original optimization level is disabled[2]. This produces a set of binaries O_p where only one transformation was omitted in each at a time. To reduce redundant work in the subsequent analysis, we apply an optimization that compares each of these custom binaries against the fully optimized one: if the `.text` section is identical to that of the standard reference optimization level, we discard it. This indicates that disabling that pass had no effect on the generated code, and therefore is unlikely to have meaningfully impacted debug information either.
2) *Trace extraction*: Given the unoptimized binary $unopt_p$, the optimized variants O_p, and the input set I_p, we generate debug traces for each binary. These traces allow us to evaluate execution halting and state retrieval—two capabilities that enable common debugging tasks such as source-line stepping and (conditional) breakpoints. We execute the program in the debugger with all the inputs in one run, setting a temporary breakpoint at each line in the DWARF line number table. When a breakpoint is hit, we record the line identity and the variables *available* at it (i.e., visible and with an associated value [4]), and resume execution until the next breakpoint with the `continue` command.

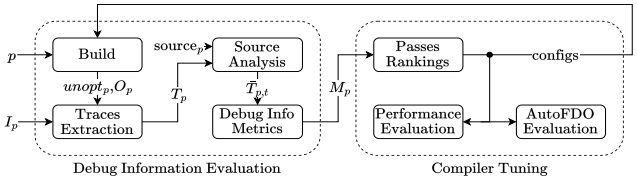

Fig. 1. The DebugTuner framework.

3) *Source Analysis*: We statically analyze the source code to determine the definition ranges of all variables in the debug traces. This allows us to refine the traces, filtering out variables that were available in the debugger but not in scope in the source, thus avoiding distortions caused by compiler-level artifacts like DWARF definitions at `O0` noted in [20]. This results in a clean trace $\bar{T}_p$.
4) *Metric Computation*: From $\bar{T}_p$, we compute two main debug information quality metrics, line coverage and variable availability, and we used those to compute the product of metrics M_p for each compiler configuration.

B. Compiler Configuration Tuning

The second component of the framework analyzes debug information quality results obtained from multiple programs to determine which optimization passes have consistent and meaningful impact on debuggability across programs.

While ranking the effects of passes on debug information for a single program would be straightforward, doing so across diverse code bases introduces additional complexity due to variability in how optimizations behave. Our approach aggregates results across multiple applications to identify patterns and provide more stable guidance for compiler tuning. In Section V-A, we discuss experimental evidence that supports this reasoning, as most passes at the top of the average ranking tend to be among the most impactful passes for each program. We underline that the same rationale is behind how optimization levels are built for performance in mainstream compilers, picking passes that tend to be beneficial to many.

Given the set M of debug quality metrics computed across all programs, we assess the impact of disabling each pass t from a standard optimization level o. For a given program, let $M_{o,t}$ be the product metric score with pass t disabled and M_o the baseline score for level o. Passes fall into three categories:

- $M_{o,t} > M_o$: disabling t improves debug quality;
- $M_{o,t} = M_o$: disabling t has no measurable effect, either because it affects only code that was not exercised or it did not alter the final binary;
- $M_{o,t} < M_o$: disabling t degrades debug quality, possibly because the pass retains useful debug information or helps subsequent passes in the optimizer to do so.

Since optimization effects are code-dependent, we should estimate each pass's impact across the entire test suite. The ranking we produce accounts for positive, negative, and lack of effects from disabled passes. We start by computing the

[2]If the level invokes the pass multiple times, all occurrences are disabled.

relative increment in the product metric from disabling the pass. For example, with an original product metric score of 0.25 and an improved one of 0.27, this increment equals to $0.02/0.25 = 0.08$. Passes with no effect get identical low rank, and those with negative impact are ranked below them. To avoid bias from outliers, we rank passes by their average rank in per-program rankings, rather than by increment magnitude.

To form the final ranking, we construct compiler configurations by exploring subsets of the top k ranked passes, with $k = 10$ in our experiments. We also compute the geometric mean across programs of the relative increment from disabling the pass to more readily estimate the average effect.

From the final ranking, we construct new compiler configurations by selecting subsets of the top 10 ranked passes. Starting from the highest-ranked pass, we incrementally build configurations by including more passes in the set of what to disable (feedback loop edge in Figure 1).

Finally, we use the selected compiler configurations to measure the performance and debuggability effects on applications relevant for end users. In the evaluation, we use the SPEC CPU 2017 benchmarks to study the performance characteristics for these debug-friendly compiler configurations, and AutoFDO to showcase the effect of improved debug information quality.

C. Implementation

We implemented DEBUGTUNER in Python and Bash, with approximately 3000 lines of code.

For building purposes, we extract the list of enabled passes at the different optimization levels of gcc and clang, parsing in an automatic fashion, specific files and outputs of the compilers. As Section V details, we manipulate passes for both the intermediate and lower-level code representations.

While gcc offers command-line switches to disable passes, we modified LLVM to support this configurability. Our patch ($\sim$100 lines) is inspired by the tool `opt-bisect-limit`: it uses an `OptPassGate` check to inspect the name of each pass about to execute and decides whether to skip it. LLVM developers recently merged our patch into the code base, with favorable comments from industry developers interested in examining pass behavior for bug fixing and troubleshooting.

For debug trace extraction, as discussed in Section III-A, we build on the prototype of Assaiante et al. [4], running a program and its inputs in the debugger, with temporary breakpoints set at each line in the DWARF line number tables the compiler emits for each compilation unit (i.e., source file). Using temporary breakpoints targets efficiency, as the reached lines are stepped only once, and the debug trace for the session is exported as a JSON file to ease offline trace comparisons.

Finally, we implement a Python tool ($\sim$400 lines) to parse the abstract syntax tree from the source and identify the assigned variables in scope for each line, needed for addressing the DWARF definition issues at `O0` mentioned in Section II.

D. Discussion

DEBUGTUNER is designed to evaluate how compiler toolchains lose debug information when optimizations are applied. It measures the impact of disabled individual optimization passes to understand their effect on debuggability, measured through one or more metrics of choice (in this study, we selected the product metric of Section II).

The design of DEBUGTUNER currently does not incorporate performance considerations during the pass selection process for improving debuggability. This means that we only identify where debug information is lost more in mainstream compiler settings optimized for performance, and estimate how much of that can be regained with minimal changes (to the pipeline) and the cost for it (from less optimization). However, optimizing for both performance and debuggability at once remains a potential research direction, and we explore the Pareto front for this trade-off in the experiments discussed in Section V-A.

DEBUGTUNER requires minimal knowledge of the compiler, namely access to control toggles for the enabled passes. This makes our framework suitable for a wide range of users: from compiler developers seeking to assess the impact of specific passes, to end users aiming to improve debuggability without significantly sacrificing performance.

Users of DEBUGTUNER should be careful about assuming effect independence of passes when interpreting the results. By that, we mean that for some passes, part of the measured improvement may not be directly attributable to the omitted pass, but rather to actions originally done by subsequent passes and that no longer occur on the program throughout the modified optimization pipeline. A prominent example of such behavior is with inlining, which we discuss later in the paper. This aspect may (e.g., for pass developers) or may not (e.g., with AutoFDO) be relevant depending on the end-user kind.

Our design remains, by construction, blind to potential inter-dependencies among multiple optimization passes if disabled at once. While we experimentally measure the benefits for sets of disabled passes derived from our ranking, we do not explore the full space of combinations because their exponential number makes the study computationally infeasible. We believe that, for general compiler research, researchers should tackle the identification of pass inter-dependencies, and such an analysis would prune the space for our method to explore.

The test suite we assemble and experiment with in the remainder of the paper is made of C code. Technically, DEBUGTUNER is compatible with any clang and gcc-supported language that sees debug information emitted in DWARF. These include C/C++, Rust, Fortran, and Objective-C. For some, our range analysis may need adaptations, as we obtain the AST from clang's frontend. We leave the analysis of other languages and of multi-language code bases to future work.

IV. A DEBUG INFORMATION TEST SUITE

This section proposes a practical approach for obtaining representative inputs for real-world applications, leveraging community efforts in continuous fuzzing campaigns to build a set of heterogeneous applications with high-coverage inputs.

Debugging real-world programs: Symbolic debugging, as a dynamic analysis technique, covers the paths exercised during execution, making input quality crucial for achieving high code coverage. In our context, this poses a significant

TABLE III
STATISTICS ON PROGRAMS AND INPUTS FOR THE TEST SUITE.

program	corpus		coverage		
	avg inputs (minimized)	% reduction	steppable lines	stepped lines	% debug coverage
bzip2	26	99.31	2099	1705	81.23
libdwarf	280	97.08	29030	15632	53.85
libexif	338	96.68	4111	2978	72.44
liblouis	41	99.32	7359	3900	53.00
libmpeg2	242	96.19	7272	5576	76.68
libpcap	690	95.24	10868	4808	44.24
libpng	280	98.04	8300	2872	34.60
libssh	53	97.39	19131	2796	14.62
libyaml	171	97.88	5271	3160	59.95
lighttpd	32	98.19	979	274	27.99
wasm3	214	96.99	2976	1734	58.27
zlib	30	98.75	3836	2049	53.42
zydis	122	96.61	5984	3567	59.61
average	194	97.51	8247	3927	48.83

challenge: to accurately evaluate debug information, we must ensure that a large portion of the program is executed.

To illustrate this, consider the scenario in which a program is executed with a single input that exercises only a narrow portion of the code base. The resulting debug information analysis would then be limited to just those executed lines, severely under-representing the true scope of the debug symbols present in the binary. Poor coverage is also more likely to fail to capture the effects of specific passes on debuggability.

To address this challenge, we present a practical method to derive a test suite with minimal effort and sizeable coverage. The method relies on input generation strategies and continuous testing efforts from the fuzzing community. We benefit from OSS-Fuzz [21], a large-scale, continuous fuzzing campaign that targets hundreds of popular open-source software by running multiple fuzzers [24], [25] on them 24/7.

We take advantage of two key features of OSS-Fuzz: (1) it provides harnesses to execute test inputs, and (2) it maintains a corpus of inputs that capture all the code coverage ever reached on the program. This allows us to sidestep the need to write test drivers and ensures that we achieve meaningful execution across diverse regions of the code bases under analysis.

Test suite construction: We demonstrate the approach by building one test suite with 13 real-world C programs tested by OSS-Fuzz. Each program includes one or more fuzzing harnesses, which allow for targeted testing of distinct code regions. We selected programs to maximize diversity, with the goal of testing a wide range of compiler behaviors while mitigating potential bias from redundancy in code structure.

Table III shows the test suite composition and some statistics. We built the programs with the default OSS-Fuzz compilation options, compiling with gcc and setting the optimization level to `O0` for the later analyses. We use all the harnesses available for each program: these average at 5 per program, with libdwarf peaking at 28 as an outlier (with one of them being buggy and excluded from the test suite). As anticipated in Section III-A, we seek to feed each harness with all the suitable inputs in one run, executing it in the debugger with temporary breakpoints placed on all the supported lines.

The corpora maintained by OSS-Fuzz are helpful to initiate new fuzzing campaigns, seeding them with the wealth of code coverage uncovered by previous campaigns. These corpora, also known as *queues*, may contain thousands of inputs for a single harness: we measure an average of 6905 inputs.

For our purpose, these inputs are overly abundant because, for example, they may differentiate executions also by the frequencies of the traversed control flow edges. To measure the availability of debug information, covering a source line once is already sufficient, whereas the frequency is irrelevant.

To accelerate subsequent analyses, we can prune the queue to obtain a subset of inputs that collectively exercise the same code as the full queue. This action is known as *minimization* in fuzzing literature. We do this code coverage-preserving minimization using a standard fuzzing facility: `afl-cmin`.

Next, we prune the queue in terms of debug trace coverage. That is, we remove inputs that, when debugging, let us step on lines already covered by other inputs. A fast approximation of the set cover problem is acceptable here: we build traces for all inputs, process those with most unique stepped lines first, and discard any input that does not introduce new stepped lines compared to those processed and retained until then.

The full pruning results in an average queue size reduction of 97.51%, with a final per-harness average of 194 inputs. Table III shows the post-minimization input counts per harness and the percentage reduction compared to the full queue.

The table also shows how these inputs cover the lines visible for the debugger. For each unoptimized binary, we extract the number of steppable lines (i.e., those associated with debug information and eligible for breakpoints) and compare this to the number of unique stepped lines observed during execution. The ratio gives a sense of how much of the available debug information is exercised by the set of inputs. This coverage averages at 48.83% and is usually high with few exceptions (mainly, 14.61% for libssh), backing the validity of the method that we use to opportunistically build a rich input test suite.

A peek at debug information quality: Table IV shows the results from computing the product metric for debug information availability on the programs in our test suite. We consider the levels `O1`, `O2`, `O3`, and for gcc also `Og`. Programs that could not be built with clang due to the usage of compiler-specific constructs and related issues are left blank.

Overall, both compilers perform better on real-world software than on synthetic programs (Section II). On gcc, the real-world programs show an average metric value that is 25.20%, 85.94%, 126.17%, and 121.80% higher than in synthetic ones at `Og`, `O1`, `O2`, and `O3`, respectively. The trend is analogous with clang, with increases of 103.61%, 106.48% and 105.25% for its three levels. As for individual programs, the only one showing a metric value lower than synthetic programs is libmpeg2 at `Og` with 0.4439 (vs. 0.4993). These numbers back our claim that synthetic programs used in prior studies are not adequate to study the debuggability of programs.

Between compilers, we observe different behaviors: in gcc, the quality of debug information decreases more sharply with higher optimization levels, while in clang, the degradation is smaller. For example, gcc's value drop from `O1` to `O2`

TABLE IV
DEBUG INFORMATION AVAILABILITY ON OUR TEST SUITE AND NUMERICAL DIFFERENCES BETWEEN COMPILERS, MEASURED FOR STANDARD OPTIMIZATION LEVELS. FOR READABILITY, VALUES ARE REPORTED TO TWO DECIMAL PLACES.

program	gcc				clang			Δ (%) gcc vs. clang		
	Og	O1	O2	O3	O1	O2	O3	O1	O2	O3
bzip2	0.71	0.64	0.41	0.40	0.57	0.49	0.48	11.47	-16.49	-15.74
libdwarf	0.65	0.56	0.43	0.41	0.41	0.39	0.39	38.29	8.83	5.74
libexif	0.53	0.50	0.35	0.33	0.50	0.49	0.49	0.00	-28.66	-32.60
liblouis	0.66	0.57	0.47	0.46	-	-	-	-	-	-
libmpeg2	0.44	0.38	0.28	0.26	-	-	-	-	-	-
libpcap	0.54	0.46	0.33	0.31	-	-	-	-	-	-
libpng	0.73	0.65	0.52	0.47	0.64	0.60	0.60	1.06	-14.51	-21.52
libssh	0.63	0.58	0.50	0.49	0.56	0.53	0.53	3.06	-6.18	-6.36
libyaml	0.69	0.57	0.47	0.47	0.54	0.52	0.51	5.86	-9.83	-8.73
lighttpd	0.64	0.58	0.41	0.41	0.56	0.55	0.56	4.16	-26.07	-27.08
wasm3	0.68	0.60	0.48	0.47	0.63	0.60	0.59	-5.22	-19.43	-19.72
zlib	0.68	0.63	0.50	0.44	0.63	0.60	0.60	-0.08	-16.38	-26.51
zydis	0.61	0.54	0.39	0.36	0.47	0.43	0.43	15.67	-9.71	-15.11
average	0.63	0.55	0.42	0.40	0.55	0.52	0.51	7.42	-13.84	-16.76

averages 0.1352, compared to just 0.03 in clang. This suggests a more robust debug metadata-preserving approach in LLVM's optimization pipeline, which shows also in the relative differences between compilers at comparable levels. At `O1`, gcc outperforms clang by 7.43% on average, but at `O2` and `O3` it produces 13.84% and 16.76% less information, respectively.

V. EXPERIMENTAL RESULTS

This section presents the experimental results obtained by evaluating our method on both clang and gcc. Our analysis is structured around the following research questions:

RQ1: Which compiler passes are most detrimental to debug information quality?

RQ2: How effective is our compiler tuning strategy in constructing debug-friendly optimizer configurations?

RQ3: Can the generated configurations improve the effectiveness of AutoFDO?

Experimental Setup: All experiments were conducted on a server equipped with an Intel Xeon E5-2699 v4 CPU, 256 GB of RAM, running Linux OpenNebula3 with kernel version 5.4.0. We used gcc 12.2.0 (commit `2ee5e4`) and clang 20.1.0 (commit `0e240b8`), being them the latest trunk versions available at the time of evaluation. As debugging tooling, we employed gdb 13.2 and lldb 20.1.0 (same commit as clang). For SPEC CPU 2017, as in recent studies on compiler optimization [26], we consider the 8 C/C++ Integer benchmarks left after excluding 520.omnetpp as it does not build with clang. Following SPEC best practices, for each benchmark we report the median execution time from running it in three separate iterations, with each iteration consisting of 20 parallel copies running on distinct physical cores. Finally, as in systems benchmarking practices, we disabled frequency scaling, minimized background activity, and pinned cores.

A. Effects of Optimizations on Debug Information

Section III-B illustrated how we compute a global ranking of passes by averaging the per-program ranks for the test suite.

As detailed in Table VII in the Appendix, each optimization level enacts passes in the order of one hundred (78-147), with about half of them being able to contribute an increment for debug information availability on the test suite if disabled.

To keep the evaluation costs tenable, we limit our analysis to the top 10 passes per optimization level, leaving to future work the investigation of strategies for considering more combinations of passes at scale (Section VI).

Although this means many passes are excluded, the selected passes provide a high coverage of the source-level elements recoverable by singularly disabling each possible pass. Specifically, on gcc, the passes considered for the evaluation collectively cover on average 71.65%, 85.38%, 74.36%, and 78.13%, of the source-level elements recoverable at `Og`, `O1`, `O2`, and `O3`. On clang, these values are even higher with 88.05%, 82.67%, and 77.82% for `O1`, `O2` and `O3` respectively. In most cases, the impact of single excluded passes is also residual, following a long-tail distribution with only minor contributions from each pass. Therefore, focusing on the top 10 passes provides a good trade-off between the cost of the experimentation and the relevance of the insights obtained.

Tables V and VI list the top 10 debug-critical passes for gcc and clang, respectively, ranked by their average position across all programs, along with the average relative increment of debug information (Section III-B), expressed as a percentage improvement on the reference optimization level. To differentiate passes, those applied by the compiler's back-end to the lower-level representation are marked with an asterisk ('*').

One clear insight from the data is that the inliner consistently ranks first, with an impact on debug information substantially higher than other passes in the rankings. However, this effect arises not directly from the inliner itself (which inlines function bodies into their callers), but rather from it enabling optimizations that disrupt debug information when the code is inlined at multiple locations. Supporting this claim, the influence of the inliner grows with increasing optimization (from 12.54% to 30.18% in gcc, and from 10.56% to 12.83% in clang). The only exception is `Og` in gcc, with inlining not in the top ranking likely due to its weakened form for the level.

Moving to the remaining passes, who instead have a direct effect on debuggability, the top positions are commonly taken by passes that substantially change code structure (e.g., block reordering). A noteworthy observation is the consistent presence of low-level optimizations across all levels and both compilers,

TABLE V

TOP 10 CRITICAL OPTIMIZATION PASSES IN GCC (PASS NAME AND PERCENTAGE IMPROVEMENT FOR DEBUG INFORMATION QUALITY), ORDERED BY AVERAGE RANK POSITION AMONG ALL PROGRAMS IN THE TEST SUITE. BACK-END PASSES ARE ANNOTATED WITH AN ASTERISK ('*').

#	Og		O1		O2		O3	
1	thread-jumps *	6.38	inline	12.54	inline	26.08	inline	30.18
2	reorder-blocks *	6.39	toplevel-reorder *	10.24	schedule-insns2 *	16.74	schedule-insns2 *	17.67
3	tree-coalesce-vars	6.22	thread-jumps *	9.28	inline-small-functions	16.47	inline-small-functions	19.88
4	tree-forwprop	6.15	inline-fncs-called-once	10.38	toplevel-reorder *	16.99	thread-jumps *	15.96
5	tree-fre	6.39	tree-sink	7.85	thread-jumps *	14.75	tree-loop-optimize	15.33
6	dce	5.99	tree-dominator-opts	8.73	crossjumping *	12.94	crossjumping *	14.22
7	guess-branch-probability	5.91	tree-loop-optimize	7.76	inline-functions	13.48	expensive-opts	14.74
8	shrink-wrap *	5.88	tree-ter	7.52	tree-loop-optimize	13.60	inline-functions	15.37
9	ipa-pure-const	5.74	tree-ch	7.27	expensive-opts	13.00	tree-slp-vectorize	13.36
10	ira-share-spill-slots *	4.63	reorder-blocks *	6.79	if-conversion	12.07	toplevel-reorder *	15.59

TABLE VI

TOP 10 CRITICAL OPTIMIZATION PASSES IN CLANG. MEANING OF COLUMNS AND '*' IS INTENDED AS IN TABLE V.

#	O1		O2		O3	
1	Inliner	10.56	Inliner	12.53	Inliner	12.83
2	SimplifyCFG	6.72	JumpThreading	4.43	Machine code sinking *	3.10
3	Machine code sinking *	2.22	Machine code sinking *	2.87	JumpThreading	4.22
4	InstCombine	2.90	SimplifyCFG	4.59	LoopStrengthReduce	2.14
5	Control Flow Optimizer *	1.30	LoopStrengthReduce	1.91	SimplifyCFG	4.33
6	EarlyCSE	2.84	Control Flow Optimizer *	1.65	Branch Prob BB Placement *	1.76
7	LoopStrengthReduce	1.46	DSE	1.61	DSE	1.67
8	Branch Prob BB Placement *	0.85	GVN	3.18	LoopUnroll	1.79
9	LoopRotate	1.03	LoopRotate	1.22	Control Flow Optimizer *	1.74
10	SROA	2.25	SROA	2.03	SROA	1.89

underscoring that debug information loss can occur in both the middle-end and the back-end of modern optimizers. This indirectly shows how DEBUGTUNER captures the impact of optimizations throughout the entire compilation pipeline.

To complement the argument of Section III-B, we now examine whether averaging over the set of programs is justified when compared with per-program rankings. The average rankings show good correspondence with program-specific results: notably, 7–8 of the passes in the presented ranking consistently appear within the top 10 passes for individual programs, while the remaining passes are consistently found within the top 20.

A further aspect worth noting is that, alongside fine-grained controls for all individual passes, gcc offers command-line toggles that control groups of passes at once. As DEBUGTUNER treats the optimizer in a black-box fashion, it sees these groups as a single independent pass for the ranking—something that may even be convenient for end users. Notably, the `expensive-optimizations` group encompasses several GIMPLE-level passes known to increase compilation time. By disabling this group, DEBUGTUNER assesses the benefits of disabling all such passes at once: this leads to a percentage improvement of 13% at `O2` and 14.74% at `O3`, making it a top-10 entry for both levels. This dynamic does not occur in clang, as our patch (Section III-C) toggles single passes only.

The main difference between the rankings of the two compilers lies in the percentage improvement values. gcc exhibits higher values than clang, particularly at higher optimization levels. This difference may stem from the fact that, as noted in the discussions for Table II and Table IV, clang tends to preserve debug information better than gcc, which ultimately results in a lower amount of recoverable source-level elements.

Another relevant difference is the inter-level variability of the top entries in the ranks, higher in gcc. In particular, `O1` and `Og` share only two passes in the top 10, which we attribute to the developers actions in disabling or weakening optimizations when constructing `Og`. Analogous differences are present between `O1` and `O2`, whereas between `O2` and `O3`—which builds incrementally on `O2`—the dynamic is inverted, with 9 passes in common (yet in different positions).

On the contrary, the transitions are more nuanced in clang, since all levels are built incrementally. Besides inlining, we identify 5 passes as recurring in all three top-10 rankings. While some loss of debug information is often inevitable, developers may consider prioritizing them for strengthening.

B. Debug-friendly Compiler Configurations

Based on the rankings presented above, we can tune the target compilers to construct configurations that improve debug information quality in the test programs for a limited performance penalty measured on SPEC CPU 2017.

For each compiler and optimization level, we evaluated several configurations constructed by disabling subsets of passes. We refer to each configuration as `Ox-dy`, where `x` is the tuned optimization level and `y` the amount of passes disabled, chosen as the top `y` of the ranking for level `x`.

The inliner transformation sees a special treatment. Intuitively, one would exclude it for two factors: first, as previously noted, the inliner is not directly responsible for debug information degradation; second, it can have a significant impact on performance. We do that for clang's *Inlining* pass.

As for gcc, it offers options that selectively disable specific inlining behaviors. We chose to exclude only the general `inline` flag and consider instead the other, more specific flags for configuration construction, since the performance

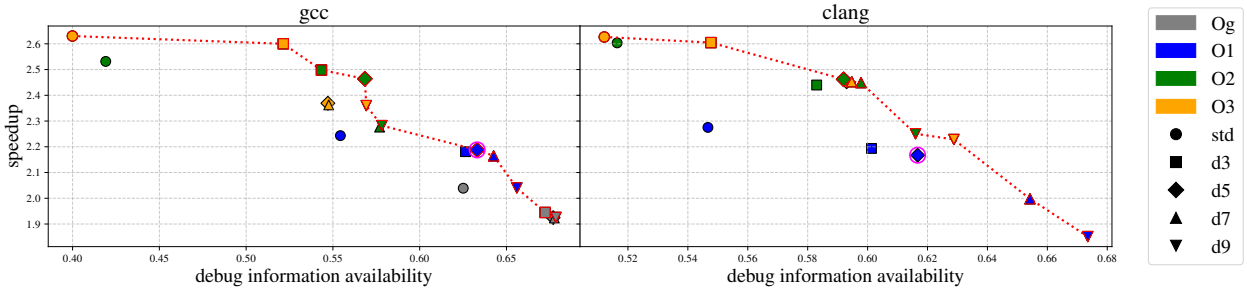

Fig. 2. Trade-off between debug information availability and performance. All optimization levels and tested configurations are included. The x axis plots the product metric defined in Section II, whereas the y axis plots the speedup over `O0`. Colors represent optimization levels, while markers distinguish specific configurations (standard vs. `dy`). The red dotted line represents the Pareto front. In *Takeaways for developers*, we refer to the configurations circled in magenta.

impact of partially disabling inlining is expected to be moderate. Analogously for debuggability, the `O3` data suggest that disabling `inline` gives a 30.18% improvement, which becomes only 15.58% with `inline-functions` (and 19.88% with `inline-small-functions`) disabled.

To explore the interplay between the running time improvement (speedup over `O0`) and the availability of debug information, we compute the Pareto front across all tested points, as depicted in Figure 2. Pareto-optimal configurations are non-dominated: no other configuration achieves both higher debug information and higher speedup simultaneously. In the appendix materials owing to space limitations, Table VIII lists the relative variations of debuggability (top) and speedup (bottom) of `Ox-dy` configurations over the `Ox` reference.

When the objective is to maximize performance alone, the `O3` configuration represents the optimal point for both compilers, appearing at the top-left of the charts in Figure 2. In contrast, configurations that gain more debuggability (`Og-d9` for gcc and `O1-d9` for clang) lie at the opposite corner. All configurations lying on the Pareto front between these extremes represent the optimal trade-offs between performance and debuggability. For gcc, 11 of 20 configurations lie on the Pareto front, while for clang this happens with 9 out of 15 (Appendix Table XIII and XIV list them in detail with values).

In gcc, the `O1-d3` and `O1-d5` configurations outperform `Og` in debug information availability (0.6264 and 0.6331 vs. 0.6251) for a superior performance (2.18x and 2.19x vs. 2.04x on the `O0` baseline—a ∼7% improvement for both vs. the `Og` baseline). We find this a strong result, suggesting our automated method may identify critical passes for debug information more effectively than developer experience alone. Unlike `O1-d5`, we note that `O1-d3` is not Pareto-optimal, but is still valuable as it lies in close proximity of the front.

We then appreciate how for both `O2` and `O3`, disabling the top 3 passes in the rankings we are able to have a large positive impact on debug information (+29.67% and +30.39%) for a rather low performance penalty (−1.33% and −1.14%). Both `O2-d3` and `O3-d3` are Pareto-optimal configurations.

With clang, disabling sets of passes typically introduces a higher performance penalty compared to gcc. Nevertheless, looking at `O1-d5`, we recover a notable amount of debug information (+12.80%) for a limited performance overhead (−4.74%). We consider such overhead limited as it is arguably smaller than the ∼10% difference in performance measured on SPEC between `Og` and `O1` in gcc, in which `Og` is a fine-tuning of `O1`. We see in `O1-d5` a valid initial candidate for deriving an `Og` level in clang—something that recently attracted interest from LLVM developers [27]. While not Pareto-optimal, we prefer it over `O2-d9` (faster for a similar debuggability) as it disables significantly fewer passes while still improving debuggability in appreciable amounts, and this may be received better, at least initially, within a mature compiler.

The data above corroborate our claim that more debug-friendly optimizing pipelines can be constructed for a reasonable performance penalty. In the hands of compiler developers and with further testing[3], DEBUGTUNER can offer a means to fine-tune and revisit the reference settings of modern compilers. While our design does not optimize simultaneously for debuggability and performance (Section III-D), the results are encouraging. Among the standard levels, only `O3` in both gcc and clang lies on the Pareto front, achieving the highest performance, with clang's `O2` performing similarly (coherently with studies on the at times limited benefits of `O3` over `O2` [28]). All other levels fall short of the front and are dominated by our configurations, reflecting the positive impact of the disabled passes on two objectives inherently in tension.

> ***Takeaways for developers.*** *For gcc, we built a more performant* `Og` *by disabling only three passes from* `O1`*: toplevel-reorder, thread-jumps, inline-fncs-called-once. Results further improve by disabling tree-sink and tree-dominator-opts. For clang, five passes cause recurring loss at all levels: SimplifyCFG, Machine code sinking, Control Flow Optimizer, LoopStrengthReduce, SROA. A prototype* `Og` *level may disable the first three from* `O1` *along with InstCombine and EarlyCSE as with* `O1-d5`*.*

C. Case Study: AutoFDO

In this section, we examine how improved debug information availability can improve downstream tools, focusing on AutoFDO as a use case. AutoFDO is a profile-guided optimization technique that uses sampled execution profiles from hardware performance counters to guide compiler decisions. It relies on

[3]For how debuggability benefits generalize and for more overhead studies.

debug information to map instruction addresses sampled when profiling such as function calls and branches back to source-level constructs. The effectiveness of AutoFDO profiles thus directly depends on the debug information that the compiler emitted for the binary used in the profiling stage.

Typically in production environments, profile collection occurs from binaries already optimized by AutoFDO: this is known as iterative or multi-round AutoFDO. In this study, we evaluate a single iteration without loss of generality to multi-round AutoFDO profiling. This choice still reflects practical use and allows us to isolate and quantify the benefits of improved debug information in a controlled context.

As most of the recent AutoFDO developments target clang, we do not test it with gcc. We expect gcc's implementation of AutoFDO to benefit even more given the sharper degradation of debug information with increasing optimization levels.

Compared to instrumentation-based FDO on an identical workload, AutoFDO incurs the risk of working on profiles that do not fully reflect program behaviors. In practice, partial information (due to sampling) or imprecision in reconstructing and processing these profiles may even lead to cases where AutoFDO binaries perform worse[4] than standard optimized binaries. Nevertheless, by improving the available debug information, our approach can directly contribute to more accurate profile reconstruction: therefore, we typically expect performance gains over standard AutoFDO binaries.

For the study, we pick the same SPEC benchmarks used for the other research questions. To better mimic a production environment, we profile them on the *ref* workload for benchmarking runs rather than the smaller *train* one. We reproduce the evaluation setup of the AutoFDO paper [16], compiling with `O2` (and our tuned configurations) for the profiling stage. Following established practices, we add the `-fdebug-info-for-profiling` flag [29], which improves the work of AutoFDO (hence, the speedup) by ensuring that the emitted DWARF information always includes, among others, the start line and linkage name of all functions.

Figure 3 presents the best AutoFDO results achieved using profiles collected with our various `O2-dy` configurations. For each benchmark, we compare the running time of the AutoFDO build from an `O2` profile with the most performant `O2-dy`-profiled build (orange bar). For context, we compare also the running time of a standard `O2` build (blue bar).

The speedup from our method, listed atop orange bars, averages at 1.47%, with effects ranging from 0.50% on 531.deepsjeng to more notable 3.67% on 505.mcf and 2.42% on 557.xz. Figures in Appendix Table XV show that all our configurations bring concrete improvements (+6.68% for `O2-d3` to 8.65% for `O2-d9` on average) in available debug information, measured by counting as a proxy the unique lines steppable when loading each benchmark in the debugger.

These experiments showed also that AutoFDO may sub-optimally use richer profiles, resulting in occasional regressions

[4]The attentive reader would notice that, in 3 out of the 8 benchmarks evaluated, the baseline AutoFDO configuration resulted in slower binaries. In the AutoFDO paper, this happened on SPEC CPU 2006 only for 525.x264.

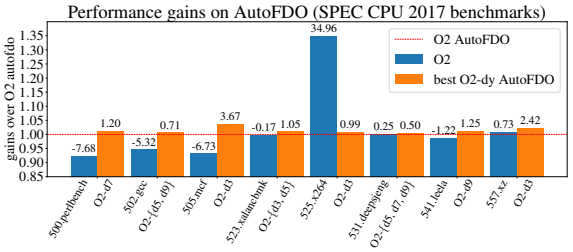

Fig. 3. Relative performance vs. `O2`-AutoFDO measured for `O2` w/o AutoFDO (blue) and our best `O2-dy`-AutoFDO (orange) on SPEC CPU 2017. Values above the bars are their percentage difference with the 1.0 reference.

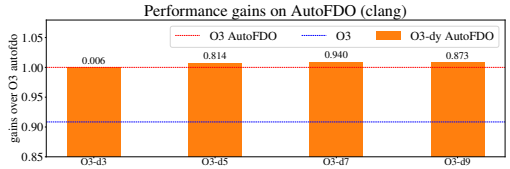

Fig. 4. Relative performance of `O3-dy`-AutoFDO vs. `O3`-AutoFDO for the self-compilation of clang 20. Values above the bars are as in Figure 3.

(Table XV). Whereas not all the extra lines we recover may meaningfully impact a profile (for example, when related to rarely executed code), we noticed some regression cases even when hot blocks or functions are added or revised in the enhanced profiles. We are currently studying these cases for reporting them to developers, and expect the general performance of AutoFDO to improve once they will be solved.

Large workloads: To further study the opportunities enabled by our approach, we test AutoFDO on a larger workload. We choose to self-compile clang, as the LLVM community often uses it as a representative workload to validate ideas before trying them in a production environment. To generate the binary for the profiling stage, we self-compile clang 20.1.0 with `-g -O3 -fdebug-info-for-profiling`, with `O3` chosen to reflect the practice mentioned above.

For the test, we profile the first 100 compilation steps of clang as in [26], using `O3` and our alternative `O3-dy` configurations. Figure 4 shows the performance gains. Compared to the setup of [26], we refine the workflow by measuring the running time for clang invocations only, leaving out linking steps since we set the LLVM build system only to profile and optimize clang (we refer to our artifacts for the details). To mitigate the effects from I/O contention, we make the 10 runs sequentially, allowing each run to use the same 10 pinned physical cores, and with the source files placed on a ramdisk.

AutoFDO with `O3` profiles ameliorates standard `O3` compilation by 10.07% (blue line at 0.09084). Our `O3-dy` configurations for the profiling stage improve the work of AutoFDO with speedups up to 0.94% (`O3-d7`) compared to when using `O3` profiles. To contextualize these numbers, as noted in prior work [26], [30], [31], even marginal gains matter at hyperscalers: given their substantial resource footprint, small improvements can translate into substantial aggregate savings. The running times are consistent across trials, with negligible variance (i.e., standard deviations peaks at 0.42 for runs taking 1250-1390 seconds across all the tested configurations).

VI. Limitations and Future Works

Fuzzing offers a cost-effective way to generate inputs with good coverage. The amount of coverage attained per program varies according to several factors, such as the quality of the available harnesses and the presence of fuzzing roadblocks in the code. This limitation is mitigated by the availability of hundreds of programs on OSS-Fuzz. While our analysis of a code base remains incomplete, it ensures that what we inspect reflects what end users work on (e.g., when debugging or for scenarios like AutoFDO), unlike static methods [20] that overestimate debug information by considering dead code and debug symbols that do not materialize when debugging.

In Section III-D, we discussed how DebugTuner reasons on one pass at a time for practicality, missing potential inter-dependencies and cascade effects. While identifying such dependencies is an open problem in literature, we believe our approach could be further tuned by using optimization algorithms. These algorithms may explore the space of passes for constructing subsets in more effective ways than considering the top rank entries as we do in this paper. We plan to tackle this direction in the future.

Another limitation concerns pass ordering. DebugTuner aims to improve existing optimizations levels in mainstream compilers, enhancing debuggability for end users. Both gcc and clang apply passes in a fixed ordering for the level, chosen by their developers based on experience and insights from performance studies. However, several authors (e.g., [32]–[35]) have proposed machine learning and search-based techniques to build alternative pass orderings for increased performance. DebugTuner could be combined with these approaches, opening a multi-dimensional search space to design new levels with different debuggability and performance trade-offs. A relevant difference we note with performance tuning scenarios is that a pass with a negative impact on debug information in one ordering will continue to lose information in every other ordering (e.g., if the pass is moved to later in the ordering), and some orderings may amplify this effect.

One aspect to further study and strengthen is the generalizability of the results from our test suite. The improvements we expect from disabling larger sets of passes may not linearly transfer to other programs. In the RQ2 and RQ3 studies, sometimes specific optimizations disabled only by the larger sets brought the more substantial improvements. This effect is known in the optimization practice: e.g., when choosing passes to use at a specific level based on the expected benefits and costs. Analyzing more assortments of OSS-Fuzz programs may confirm our top passes and find other harmful ones.

We acknowledge the following threats to validity. For internal validity, debugger bugs can result in incomplete or inaccurate program state being shown, affecting the computed metrics for debug information availability. Recent efforts [36], [37] target these bugs. Next, the issues we identified in how AutoFDO reconstructs and uses richer profiles may make the measured performance gains an underestimation of the benefits. These issues do not affect the soundness of the study, and fixing them is an orthogonal problem. For external validity, we mentioned how the identification of lossy passes is affected by the code coverage achieved for the test programs, which depends on the fuzzing-derived test inputs we use. Our data indicates good coverage, a fact that mitigates this threat.

VII. Related Works

Debug Information Errors: The works by Li et al. [1], Di Luna et al. [2], and Assaiante et al. [4] analyzed debug traces to identify when compilers fail to update debug information accurately. For correctness bugs, the comparison of unoptimized and optimized versions of the same synthetic program was used in [1], [2]. For completeness bugs, due to the lack of a reference oracle, a conjecture-based approach was adopted in [4]. Our work shares a similar reliance on debug traces for computing quality metrics and identifying critical optimization passes to design optimal compiler pipelines. However, our focus is on understanding where debug information is lost and prioritize these passes for developer inspection, rather than diagnosing specific bugs in compilers or debuggers.

Debug Information Quality: Section II discussed previous efforts [4], [20] to measure how much debug information is available in a binary, discussing the pitfalls of using synthetic programs and the accuracy issues of dynamic and static methods employed so far in evaluations. Our work rectifies these issues and calls for attention to focus on real-world applications, since synthetic programs, albeit easier to test, fail to capture the dynamics behind debugging real-world code.

Another related study by Wang et al. [38] proposed metrics to capture the quantity and diversity of DWARF entries, focusing on bugs such as miscompilation and crashes. However, these metrics primarily aimed to measure the volume and heterogeneity of debug information rather than its quality for debugging end users. As a result, their metrics are not directly applicable to our goal of improving the debug experience for both human developers and downstream tasks like AutoFDO.

VIII. Conclusion

We presented DebugTuner, a framework for measuring debug information quality in optimized binaries. By analyzing the effects of individual passes, we identified those that most harm debuggability and demonstrated how disabling selected passes can yield debug-friendly configurations with limited performance loss. Our results show that better debug information can enhance tools like AutoFDO, suggesting practical benefits for both developers and compiler designers.

Acknowledgments

We thank our anonymous reviewers for their feedback, Mircea Trofin for many valuable discussions that helped shape this research, and David Li and Teresa Johnson for their feedback on the paper draft. This work was supported in part by a Google Research Gift and by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through project SERICS (PE00000014).

Data Availability Statement

This paper comes with a published artifact, which the evaluators from the Artifact Evaluation Committee of the conference acknowledged as both *Available* and *Reusable*. We make available the DebugTuner release as reviewed on Zenodo with DOI 10.5281/zenodo.17865056 [39], and its latest version at https://github.com/Sap4Sec/debugtuner. We are grateful to our anonymous evaluators for their feedback.

Artifact Appendix

A. Abstract

The artifact contains DebugTuner, a framework for tuning compilers towards the generation of more debuggable programs for a low performance overhead. DebugTuner is made of two components: the first analyzes the effect of compiler optimization passes on debug information availability, and the second ranks and selects passes to build improved compiler configurations. The test suite we used to evaluate the framework is made of 13 real-world programs.

The artifact is mostly self-contained and includes all the necessary data sets, dependencies, and installation scripts for the tested programs. The only external requirement that we cannot provide is the SPEC CPU 2017 benchmark suite.

B. Artifact check-list (meta-information)

- **Program:** The test suite construction scripts (Section IV) and the source code of DebugTuner.
- **Data set:** 13 real-world programs (downloaded and built only if the entire workload is executed, otherwise only 2 are used), the minimized input corpus used in the debugin information availability evaluation, and the performance evaluation scripts.
- **Run-time environment:** x86-64 Linux.
- **Hardware:** x86-64 platform, preferably with 20 or more cores. Using fewer cores increases the time estimates made throughout this appendix. For the AutoFDO experiments, a CPU with LBR (Last Branch Records) facilities is required.
- **Execution:** Scripts are provided to construct the test suite, evaluate the debug information, and set up (optional) performance experiments.
- **Experiments:** (1) Minimal working example to construct custom compiler configurations. (2) Debug information evaluation of the generated configurations.
- **How much disk space required (approximately)?:** About 11GB with minimal test suite, otherwise about 70GB.
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes with the Docker image available on Docker Hub, otherwise 3 to 4 hours if 20 cores are available for a local build.
- **How much time is needed to complete experiments (approximately)?:** 2 to 3 hours if 20 cores are available and the provided input corpus is used.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0.
- **Archived (provide DOI)?:** 10.5281/zenodo.17865056

C. Description

How delivered: The artifact is available as a self-contained archive on Zenodo [39] or alternatively from https://github.com/Sap4Sec/debugtuner.

Hardware dependencies: The artifact must be executed on an x86-64 platform, preferably with 20 or more cores to reduce the execution time. The performance evaluation, for which we only provide scripts because of the SPEC CPU 2017 availability requirement, is configured to use the exact specifications described in Section V. Therefore, 20 cores are required for those experiments.

There are no strict RAM requirements for the whole framework. However, in order to run the large workload experiments for AutoFDO, at least 32GB of RAM are required.

The Docker image requires about 6GB of storage, and the experiments require an additional 5GB.

Software dependencies: We recommend setting up the experiments using the provided Docker image, as it represents a complete environment with all the required dependencies automatically installed, including the compiler versions used in the paper's evaluation. For manual setups, kindly note that the commands used to configure the Docker image can be extracted and run as standalone commands.

Data sets: We provide the minimized input corpus used to evaluate debug information for real-world programs that use fuzzing targets from OSS-Fuzz. These files can be used to replicate the debug information availability results and to reduce the artifact evaluation time, since downloading and minimizing a new corpus would take a significant amount of time. Additionally, we provide the scripts used for the performance evaluation of the custom configurations obtained with DebugTuner.

D. Installation

The installation consists of setting up the Docker image based on Ubuntu 20.04. We recommend using the version we made available on Docker Hub, which can be pulled with the following command.

```
$ docker pull \
cristianassaiante/debugtuner:cgo26-ae
```

Eventually, to build the image locally, we provide a bash script. The `-j` option sets the number of cores used to build all the tools during the Docker build.

```
$ cd <path>/debugtuner
$ ./build.sh -j <N>
```

Next, we create the Docker container from the image using the provided script. This script creates directories in the host system to guarantee data persistence during the experiments executed inside the container. It also prepares the corpus directory using the provided data set.

```
$ cd <path>/debugtuner
$ ./run.sh
```

Finally, we enter the Docker environment using the `docker attach debugtuner-cont` command, which gives us a shell inside the artifact directory mounted in the container.

E. Experiment workflow

Important: The workflow demonstrated below uses a reduced test suite. The listed commands are configured to use the existing data set. To construct a new data set from the current OSS-Fuzz queues, replace the list of stages in the command-line options with `--all-stages`.

(E1) Run DEBUGTUNER *on a minimal test suite and construct rankings:* The first experiment involves the execution of the full DEBUGTUNER pipeline using the minimal test suite, which consists of `wasm3` and `zydis`, with both clang and gcc. To run the pipeline, execute the following command twice. Use gcc for one execution and clang for the other execution. Also, specify the number of cores[5] to be used for parallel trace computation using the `--proc` flag.

```
python3 debugtuner.py --minimal \
--stages build traces static \
metrics rankings performance \
--compiler <gcc/clang> --proc <N>
```

The "performance" step does not run the performance evaluation. Instead, it constructs a set of scripts that can be used to eventually run the evaluation without manual preparation.

(E2) Run DEBUGTUNER *to test debug information with custom configurations:* The second experiment involves evaluating the custom configurations that are constructed from the rankings produced by the previous experiment, as described in Section V. This experiment has two steps. First, run the following command, which upon completion prints the command that must be executed to test all custom configurations. Then, copy and execute such printed command. The printed command is similar to the command used in (E1) but includes the custom configurations.

```
python3 \
post-processing/get_configs_cmd.py \
--minimal --targets dt-targets \
--compiler <gcc/clang>
```

F. Evaluation and expected result

(E1) Rankings construction: Running experiment (E1) constructs rankings using the minimal test suite. The following command generates LaTeX tables that are similar to Table V and Table VI. Since the reduced test suite is used, the ranking results may differ from those in Section V. However, most of the passes included in the evaluation table should still appear in the rankings for the minimal test suite.

```
python3 \
post-processing/prettify_ranks.py \
--targets dt-targets \
--compiler <gcc/clang>
```

[5]The time estimates we give are for N=20. We recommend assigning at least N=4 to keep the experiment duration within 12 hours.

(E2) Debug information quality: Running experiment (E2) produces results regarding the debug information quality obtained by tuning the compiler based on the rankings from experiment (E1). With the following command, it is possible to inspect the scores obtained using the minimal test suite.

```
python3 \
post-processing/prettify_configs.py \
--targets dt-targets \
--compiler <gcc/clang>
```

G. Experiment customization

Full test suite experiments: To evaluate the full test suite described in Section IV and to replicate the experimental results regarding debug information, remove the `--minimal` argument when it is present in the commands. Additionally, for those willing to replicate the performance evaluation and with SPEC CPU 2017 in their availability, please refer to the README file in the artifact repository.

Adding new test programs: The README file of the artifact describes the steps required to add new programs to the test suite. The description comprises both OSS-Fuzz programs and other real-world applications, with instruction on how to structure the input corpus for the latter.

Testing multiple compiler versions: The README file of the artifact provides instructions to test multiple versions of the currently supported compilers, clang and gcc. The additional versions can be both installed system-wide or built from source.

H. Notes

Troubleshooting: We expect the artifact scripts to execute without errors. However, we experienced some issues during the building process of the Docker image (using the publicly available Docker image would avoid this issue). Since the build process clones the repositories of both gcc and LLVM, the `git clone` command may fail if the internet connection is not stable. Restarting the Docker build usually fixes this issue.

Also, we are aware of an issue rarely occurring while executing the debugger for debug traces extraction with multiple cores. Running the pipeline using less cores, or more drastically a single one, should avoid any issue.

Log Inspection: If errors occur during DEBUGTUNER execution, all the logs are stored in the `dt-log` directory. It contains a log file for each stage executed on every tested program. Each log file follows the naming format: `<stage-name>-<compiler>-<timestamp>.log`.
For more detailed output, the `--debug` option can be added to the commands. This enables, at each stage, additional diagnostic information that may help with troubleshooting.

I. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] Y. Li, S. Ding, Q. Zhang, and D. Italiano, "Debug information validation for optimized code," in *Proc. of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1052–1065. [Online]. Available: https://doi.org/10.1145/3385412.3386020

[2] G. A. Di Luna, D. Italiano, L. Massarelli, S. Österlund, C. Giuffrida, and L. Querzoni, "Who's debugging the debuggers? exposing debug information bugs in optimized binaries," in *Proc. of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1034–1045. [Online]. Available: https://doi.org/10.1145/3445814.3446695

[3] F. Artuso, G. A. Di Luna, and L. Querzoni, "Debugging debug information with neural networks," *IEEE Access*, vol. 10, pp. 54 136–54 148, 2022.

[4] C. Assaiante, D. C. D'Elia, G. A. Di Luna, and L. Querzoni, "Where did my variable go? poking holes in incomplete debug information," in *Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 935–947. [Online]. Available: https://doi.org/10.1145/3575693.3575720

[5] J. Hennessy, "Symbolic debugging of optimized code," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 323–344, jul 1982. [Online]. Available: https://doi.org/10.1145/357172.357173

[6] G. Brooks, G. J. Hansen, and S. Simmons, "A new approach to debugging optimized code," in *Proc. of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92. Association for Computing Machinery, 1992, p. 1–11. [Online]. Available: https://doi.org/10.1145/143095.143108

[7] A.-R. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code," in *Proc. of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. Association for Computing Machinery, 1996, p. 33–43. [Online]. Available: https://doi.org/10.1145/231379.231388

[8] S. Kell and J. Stinnett, "Source-level debugging of compiler-optimised code: Ill-posed, but not impossible," in *Proc. of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, aug 2024.

[9] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[10] C. Jia and W. K. Chan, "Which compiler optimization options should i use for detecting data races in multithreaded programs?" in *2013 8th International Workshop on Automation of Software Test (AST)*, 2013, pp. 53–56.

[11] M. Copperman, "Debugging optimized code without being misled," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 387–427, may 1994. [Online]. Available: https://doi.org/10.1145/177492.177517

[12] GCC contributors, "Prevent tree-ssa-dce.c from deleting stores at -Og," July 2019, gitHub commit f33b9c4. Accessed: 2025-05-08. [Online]. Available: https://github.com/gcc-mirror/gcc/commit/f33b9c40b97f6f8a72ee370068ad81e33d71434e

[13] ——, "Prevent -Og from deleting stores to write-only variables," July 2019, gitHub commit ec8ac26. Accessed: 2025-05-08. [Online]. Available: https://github.com/gcc-mirror/gcc/commit/ec8ac265ff21fb379ac072848561a91e4990c47f

[14] ——, "tame IPA optimizations at -Og," Jan 2022, gitHub commit e89b2a2. Accessed: 2025-05-08. [Online]. Available: https://github.com/gcc-mirror/gcc/commit/e89b2a270d31d7298d516ae545e256645992c7b9

[15] ——, "Adjust max-combine-insns to 2 for -Og," September 2014, gitHub commit b30e733. Accessed: 2025-05-08. [Online]. Available: https://github.com/gcc-mirror/gcc/commit/b30e733a13c9eb196b7dfbf7afd4135d7d4c4fd0

[16] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proc. of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–23. [Online]. Available: https://doi.org/10.1145/2854038.2854044

[17] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 529–540. [Online]. Available: https://doi.org/10.1145/2976749.2978340

[18] D. Zeng and G. Tan, "From debugging-information based binary-level type inference to cfg generation," in *Proc. of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 366–376. [Online]. Available: https://doi.org/10.1145/3176258.3176309

[19] L. Hafkemeyer, J. Starink, and A. Continella, "Divak: Non-invasive characterization of out-of-bounds write vulnerabilities," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, D. Gruss, F. Maggi, M. Fischer, and M. Carminati, Eds. Cham: pringer Nature Switzerland, 2023, pp. 211–232.

[20] J. R. Stinnett and S. Kell, "Accurate coverage metrics for compiler-generated debugging information," in *Proc. of the 33rd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 126–136. [Online]. Available: https://doi.org/10.1145/3640537.3641578

[21] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software," in *26th USENIX Security Symposium (Invited talk)*. Vancouver, BC: USENIX Association, Aug. 2017.

[22] D. Committee. Dwarf standard format. [Online]. Available: https://dwarfstd.org/

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: combining incremental steps of fuzzing research," in *Proc. of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[25] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proc. of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[26] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li, "Propeller: A profile guided, relinking optimizer for warehouse-scale applications," in *Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 617–631. [Online]. Available: https://doi.org/10.1145/3575693.3575727

[27] S. Tozer. Redefine og/o1 and add a new level of og. [Online]. Available: https://discourse.llvm.org/t/rfc-redefine-og-o1-and-add-a-new-level-of-og/72850

[28] C. Curtsinger and E. D. Berger, "Stabilizer: statistically sound performance evaluation," in *Proc. of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 219–228. [Online]. Available: https://doi.org/10.1145/2451116.2451141

[29] D. Chen. Add -fdebug-info-for-profiling to emit more debug info for sample pgo profile collection. [Online]. Available: https://reviews.llvm.org/D25435

[30] M. Panchenko, R. Auler, L. Sakka, and G. Ottoni, "Lightning bolt: powerful, fast, and scalable binary optimization," in *Proc. of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 119–130. [Online]. Available: https://doi.org/10.1145/3446804.3446843

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

[32] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 147–162. [Online]. Available: https://doi.org/10.1145/2384616.2384628

[33] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Sep. 2017. [Online]. Available: https://doi.org/10.1145/3124452

[34] S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. Upadrasta, "Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 121–131.

[35] J. Zhao, C. Xia, and Z. Wang, "Leveraging compilation statistics for compiler phase ordering," in *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025, pp. 533–545.

[36] H. Lu, Z. Liu, S. Wang, and F. Zhang, "Dtd: Comprehensive and scalable testing for debuggers," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643779

[37] Y. Yang, M. Sun, J. Wu, Q. Li, and Y. Zhou, "Debugger toolchain validation via cross-level debugging," in *Proc. of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 280–294. [Online]. Available: https://doi.org/10.1145/3669940.3707271

[38] T. L. Wang, Y. Tian, Y. Dong, Z. Xu, and C. Sun, "Compilation consistency modulo debug information," in *Proc. of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 146–158. [Online]. Available: https://doi.org/10.1145/3575693.3575740

[39] C. Assaiante, S. Di Biasio, S. Kumar, G. A. Di Luna, D. C. D'Elia, and L. Querzoni, "Artifact for 'Towards threading the needle of debuggable optimized binaries' paper," Dec. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17865056

APPENDIX
ADDITIONAL EVALUATION MATERIALS

Breakdown of passes: As discussed in Section III-C, we identified the passes enabled at each optimization level by parsing files and outputs produced by the compilers. Table VII shows the number of controllable passes for each optimization level and compiler, and the how many of these passes have an average positive, neutral, or negative impact on debug information quality on the programs from the test suite.

The amount of passes with negative impact is consistently low in both compilers, while the number of those with neutral impact is higher. Instead, the number of passes that positively impact quality scores increases linearly with the more numerous enabled transformations at higher levels.

A closer look at debug-friendly configurations: We provide here the complete debug information quality scores for each program in our test suite. These measurements reflect how effective the rank-based custom compiler configurations are at improving debug information availability on each tested program. Table IX presents the results for gcc, while Table X shows the corresponding values for clang.

Looking at per-program data, for the `O1-dy` configurations that on average provide better debuggability and performance than the standard `Og`, this does not hold for all the programs in the test suite. For debuggability, this happens for 6 of them at `O1-d3` and `O1-d5`, 3 at `O1-d7` and only 1 at `O1-d9`, with libdwarf being the only project for which no configuration is able to reach a better debuggability than `Og`. However, on the other subjects, we still get values extremely close to the `Og` baseline for a much better performance. For instance, with `O1-d5`, the average distance from `Og` when considering only the programs with lower scores is 0.01, but the performance improvement remains at $\sim 7\%$ (as mentioned in Section V-B). Therefore, our claim on general better debuggability and performance remains correct after validation on single programs.

For performance aspects, we report the speedup achieved by each compiler configuration on the SPEC CPU 2017 benchmarks. Table XI shows absolute speedup values using the unoptimized binary as the baseline. Table XII highlights the percentage improvement of each custom configuration relative to the optimization level we derive it from.

One notable fact—masked in the average results discussed in Section V—is that disabling specific passes can sometimes lead to performance improvements. Table XII shows some configurations yielding performance gains. For instance, in *500.perlbench*, both gcc and clang show improvements with `O1-d3`: $+0.38\%$ and $+0.18\%$, respectively; additionally, gcc sees further gains with `O1-d5` ($+1.92\%$). Another example is *505.mcf* under gcc, on which almost all our configurations (except for `Og-dy` ones) outperform their reference levels.

Finally, the programs most negatively affected in performance by the disabled passes are 525.x264 (with consistently large losses peaking at -48.65% with `O3-d9` on gcc) and 541.leela (peaking at -20.6% with `O3-d9` on clang). Both programs present similar trends with both compilers.

TABLE VII
TESTED PASSES FOR EACH OPTIMIZATION LEVEL AND COMPILER (AVERAGE AMOUNT OF PASSES WITH POSITIVE, EQUAL OR NEGATIVE DEBUG INFORMATION QUALITY IMPROVEMENTS).

compiler	optimization level			
	Og (>,=,<)	O1 (>,=,<)	O2 (>,=,<)	O3 (>,=,<)
gcc	78 (33,44,1)	91 (52,38,1)	135 (89,44,2)	147 (93,52,2)
clang	-	98 (35,57,6)	109 (46,57,6)	112 (48,58,8)

TABLE VIII
SPEC CPU 2017: PERCENTAGE IMPROVEMENT OF DEBUG INFORMATION AVAILABILITY (TOP TABLE) AND PERCENTAGE SPEEDUP REDUCTION (BOTTOM TABLE) FOR `Ox-dy` COMPILER CONFIGURATIONS.

compiler	configuration	Δ for debug inform. availability (%)			
		Og	O1	O2	O3
gcc	Ox-d3	7.56	13.01	29.67	30.39
	Ox-d5	8.29	14.22	35.63	36.81
	Ox-d7	8.41	15.92	37.66	36.96
	Ox-d9	8.55	18.33	37.96	42.35
clang	Ox-d3	-	10.00	12.91	6.95
	Ox-d5	-	12.80	14.64	15.80
	Ox-d7	-	19.67	15.76	16.14
	Ox-d9	-	23.20	19.29	22.79

compiler	configuration	Δ for speedup reduction (%)			
		Og	O1	O2	O3
gcc	Ox-d3	-4.62	-2.77	-1.33	-1.14
	Ox-d5	-5.60	-2.47	-2.67	-9.92
	Ox-d7	-5.68	-3.50	-10.08	-10.17
	Ox-d9	-5.56	-9.05	-9.86	-10.28
clang	Ox-d3	-	-3.59	-6.29	-0.83
	Ox-d5	-	-4.74	-5.43	-6.62
	Ox-d7	-	-12.18	-5.93	-6.61
	Ox-d9	-	-18.64	-13.56	-15.16

TABLE IX

DEBUG INFORMATION QUALITY ON OUR TEST SUITE: PER-PROGRAM RESULTS FOR GCC Ox−dy CONFIGURATIONS.

program	Ox-d3				Ox-d5				Ox-d7				Ox-d9			
	Og	O1	O2	O3	Og	O1	O2	O3	Og	O1	O2	O3	Og	O1	O2	O3
bzip2	0.7685	0.6987	0.4956	0.4830	0.7813	0.7010	0.5131	0.5264	0.7784	0.7006	0.5369	0.5221	0.7784	0.7464	0.5382	0.5382
libdwarf	0.6509	0.6229	0.5193	0.4856	0.6641	0.6269	0.5520	0.5125	0.6650	0.6367	0.5557	0.5151	0.6656	0.6440	0.5586	0.5597
libexif	0.5701	0.5359	0.4800	0.4727	0.5718	0.5392	0.4979	0.4879	0.5797	0.5490	0.5010	0.4884	0.5797	0.5642	0.4990	0.4994
liblouis	0.6894	0.6489	0.5735	0.5572	0.7019	0.6509	0.6053	0.5843	0.7008	0.6602	0.6173	0.5884	0.7037	0.6794	0.6203	0.6163
libmpeg2	0.4663	0.4100	0.3465	0.3347	0.4735	0.4196	0.3547	0.3629	0.4746	0.4521	0.3746	0.3635	0.4755	0.4564	0.3755	0.3816
libpcap	0.6320	0.5832	0.4955	0.4890	0.6379	0.6049	0.5244	0.5064	0.6374	0.6137	0.5297	0.5083	0.6382	0.6203	0.5297	0.5231
libpng	0.7753	0.7277	0.6482	0.6329	0.7779	0.7384	0.6742	0.6659	0.7774	0.7541	0.6907	0.6670	0.7785	0.7673	0.6958	0.6950
libssh	0.6915	0.6669	0.6087	0.6160	0.6891	0.6658	0.6417	0.6307	0.6881	0.6732	0.6435	0.6283	0.6887	0.6846	0.6445	0.6400
libyaml	0.7391	0.6941	0.6114	0.6020	0.7374	0.6960	0.6456	0.6256	0.7364	0.6966	0.6477	0.6285	0.7382	0.7119	0.6439	0.6499
lighttpd	0.6733	0.6474	0.5976	0.4760	0.6714	0.6529	0.6072	0.4975	0.6745	0.6533	0.6072	0.5034	0.6745	0.6607	0.6068	0.4975
wasm3	0.7410	0.6684	0.6155	0.6118	0.7401	0.6727	0.6294	0.6194	0.7407	0.6897	0.6340	0.6171	0.7428	0.7007	0.6380	0.6338
zlib	0.7200	0.6691	0.5982	0.5737	0.7229	0.6767	0.6304	0.6359	0.7222	0.6732	0.6382	0.6256	0.7230	0.6868	0.6310	0.6292
zydis	0.6959	0.6480	0.5580	0.5304	0.7009	0.6599	0.6048	0.5402	0.7026	0.6618	0.6056	0.5449	0.7028	0.6713	0.6179	0.6203
average	0.6723	0.6264	0.5434	0.5214	0.6769	0.6331	0.5684	0.5471	0.6776	0.6426	0.5769	0.5476	0.6785	0.6560	0.5782	0.5692

TABLE X

DEBUG INFORMATION QUALITY ON OUR TEST SUITE: PER-PROGRAM RESULTS FOR CLANG Ox−dy CONFIGURATIONS.

program	Ox-d3			Ox-d5			Ox-d7			Ox-d9		
	O1	O2	O3	O1	O2	O3	O1	O2	O3	O1	O2	O3
bzip2	0.6116	0.5836	0.5609	0.6183	0.5942	0.5859	0.6246	0.6113	0.5961	0.6472	0.6243	0.5990
libdwarf	0.4762	0.4635	0.4154	0.4868	0.4691	0.4671	0.5385	0.4756	0.4734	0.5626	0.4935	0.5198
libexif	0.5462	0.5272	0.5104	0.5504	0.5420	0.5408	0.5985	0.5446	0.5427	0.6196	0.5733	0.5831
libpng	0.7228	0.6985	0.6364	0.7372	0.7094	0.6980	0.7719	0.7192	0.6995	0.7796	0.7225	0.7185
libssh	0.5844	0.5726	0.5481	0.5903	0.5767	0.5786	0.6238	0.5855	0.5832	0.6402	0.6014	0.6085
libyaml	0.6122	0.5941	0.5454	0.6238	0.6001	0.6080	0.6857	0.5979	0.6105	0.7060	0.6068	0.6643
lighttpd	0.6227	0.6147	0.5931	0.6499	0.6196	0.6288	0.6769	0.6250	0.6288	0.6861	0.6379	0.6788
wasm3	0.6594	0.6407	0.6132	0.6780	0.6446	0.6397	0.6855	0.6465	0.6401	0.7036	0.6679	0.6427
zlib	0.6599	0.6542	0.6255	0.6812	0.6593	0.6622	0.6897	0.6772	0.6687	0.7181	0.6946	0.6654
zydis	0.5546	0.5197	0.4716	0.5891	0.5411	0.5553	0.6747	0.5338	0.5393	0.6966	0.5705	0.6312
average	0.6013	0.5830	0.5477	0.6167	0.5920	0.5930	0.6542	0.5978	0.5947	0.6735	0.6160	0.6288

TABLE XI

PERFORMANCE EVALUATION OF STANDARD AND Ox−dy CONFIGURATIONS ON SPEC CPU 2017: SPEEDUP COMPUTED OVER O0 BINARIES.

benchmark	opt. level	gcc					clang				
		standard	Ox-d3	Ox-d5	Ox-d7	Ox-d9	standard	Ox-d3	Ox-d5	Ox-d7	Ox-d9
500.perlbench	Og	1.5586	1.5070	1.4513	1.4837	1.4991	-	-	-	-	-
	O1	1.6321	1.6383	1.6635	1.6229	1.5310	1.6339	1.6369	1.5913	1.4055	1.3436
	O2	1.7581	1.7265	1.6603	1.6961	1.6796	1.6728	1.5614	1.6195	1.5969	1.6081
	O3	1.7546	1.7440	1.7653	1.7546	1.7689	1.6789	1.6606	1.6109	1.6166	1.4319
502.gcc	Og	1.5868	1.5472	1.5336	1.5228	1.5175	-	-	-	-	-
	O1	1.8829	1.8507	1.8351	1.8159	1.7714	1.9130	1.8723	1.8526	1.6449	1.5466
	O2	1.9550	1.9418	1.8993	1.8952	1.9119	1.9512	1.8884	1.9048	1.8925	1.8605
	O3	1.9817	1.9462	1.9638	1.9594	1.9462	1.9865	1.9214	1.8966	1.8925	1.6891
505.mcf	Og	1.3588	1.3700	1.3551	1.3280	1.3245	-	-	-	-	-
	O1	1.3478	1.3496	1.4028	1.4128	1.3989	1.4540	1.4500	1.3846	1.3506	1.3541
	O2	1.3387	1.3459	1.3551	1.3441	1.3795	1.3773	1.3350	1.3506	1.3541	1.3282
	O3	1.3245	1.3514	1.3588	1.3423	1.3569	1.3883	1.4185	1.3791	1.3755	1.3419
523.xalancbmk	Og	2.8309	2.7143	2.6395	2.6395	2.6662	-	-	-	-	-
	O1	3.0788	2.9915	2.9915	3.0043	2.9211	3.4777	3.3626	3.3350	3.0134	2.7008
	O2	3.5685	3.5746	3.4293	3.4975	3.4688	3.5078	3.3794	3.3963	3.3850	3.3570
	O3	3.6113	3.5625	3.4688	3.5033	3.4975	3.5445	3.5017	3.2917	3.2971	3.1008
525.x264	Og	2.7838	2.1511	2.2277	2.3161	2.3161	-	-	-	-	-
	O1	2.9281	2.9983	2.9830	2.9281	2.2221	2.7231	2.6230	2.6399	2.6188	1.9736
	O2	5.1500	4.9886	4.9463	2.7927	2.8425	7.2655	6.4392	6.4646	6.3891	3.5541
	O3	6.5581	6.4139	3.4000	3.3868	3.3673	7.2335	7.3304	6.4646	6.3154	5.1798
531.deepsjeng	Og	1.7778	1.7413	1.7131	1.6994	1.7028	-	-	-	-	-
	O1	2.0483	1.9584	1.9317	1.8678	1.8158	1.9681	1.8167	1.7805	1.6660	1.6060
	O2	2.2257	2.1360	2.1253	1.9813	1.9539	1.9926	1.8894	1.8718	1.8762	1.8045
	O3	2.1856	2.1688	1.9953	1.9906	1.9767	1.9975	1.9926	1.8545	1.8850	1.7269
541.leela	Og	3.4098	3.4315	3.3548	3.3219	3.3016	-	-	-	-	-
	O1	4.1858	3.7143	3.6889	3.6940	3.5957	4.1452	3.7478	3.7756	3.2957	3.0950
	O2	4.4111	4.3333	4.3403	4.2583	4.2250	4.4268	3.9032	3.8853	3.8676	3.7700
	O3	4.7439	4.5217	4.3333	4.3126	4.3195	4.5456	4.4268	3.9395	3.9274	3.6094
557.xz	Og	1.8612	1.8369	1.8489	1.8074	1.8074	-	-	-	-	-
	O1	2.0036	1.9182	1.9281	1.9085	1.8673	2.0447	2.0088	2.0159	1.8768	1.8495
	O2	2.0325	2.0362	2.0325	2.0252	2.0399	2.0896	1.9878	2.0302	2.0159	2.0411
	O3	2.0252	2.0325	2.0661	2.0661	2.0473	2.1011	2.0669	1.9983	2.0194	1.8707

TABLE XII
PERFORMANCE EVALUATION OF Ox-dy CONFIGURATIONS ON SPEC CPU 2017: PERCENTAGE IMPROVEMENT COMPUTED OVER REFERENCE LEVEL.

benchmark	opt. level	gcc				clang			
		Ox-d3	Ox-d5	Ox-d7	Ox-d9	Ox-d3	Ox-d5	Ox-d7	Ox-d9
500.perlbench	Og	-3.31	-6.88	-4.80	-3.81	-	-	-	-
	O1	0.38	1.92	-0.56	-6.19	0.18	-2.61	-13.98	-17.77
	O2	-1.80	-5.57	-3.53	-4.47	-6.66	-3.19	-4.54	-3.87
	O3	-0.60	0.61	0.00	0.82	-1.09	-4.05	-3.71	-14.71
502.gcc	Og	-2.50	-3.36	-4.04	-4.37	-	-	-	-
	O1	-1.71	-2.54	-3.56	-5.92	-2.13	-3.16	-14.02	-19.16
	O2	-0.67	-2.84	-3.06	-2.20	-3.22	-2.38	-3.01	-4.65
	O3	-1.79	-0.90	-1.13	-1.79	-3.28	-4.53	-4.73	-14.97
505.mcf	Og	0.83	-0.27	-2.27	-2.53	-	-	-	-
	O1	0.14	4.08	4.82	3.79	-0.28	-4.77	-7.12	-6.87
	O2	0.54	1.22	0.40	3.05	-3.07	-1.94	-1.69	-3.56
	O3	2.04	2.59	1.35	2.45	2.17	-0.66	-0.92	-3.34
523.xalancbmk	Og	-4.12	-6.76	-6.76	-5.82	-	-	-	-
	O1	-2.84	-2.84	-2.42	-5.12	-3.31	-4.11	-13.35	-22.34
	O2	0.17	-3.90	-1.99	-2.80	-3.66	-3.18	-3.50	-4.30
	O3	-1.35	-3.95	-2.99	-3.15	-1.21	-7.13	-6.98	-12.52
525.x264	Og	-22.73	-19.97	-16.80	-16.80	-	-	-	-
	O1	2.40	1.87	0.00	-24.11	-3.67	-3.05	-3.83	-27.52
	O2	-3.13	-3.95	-45.77	-44.81	-11.37	-11.02	-12.06	-51.08
	O3	-2.20	-48.16	-48.36	-48.65	1.34	-10.63	-12.69	-28.39
531.deepsjeng	Og	-2.05	-3.64	-4.41	-4.22	-	-	-	-
	O1	-4.39	-5.69	-8.81	-11.35	-7.69	-9.53	-15.35	-18.40
	O2	-4.03	-4.51	-10.98	-12.21	-5.18	-6.06	-5.84	-9.44
	O3	-0.77	-8.71	-8.92	-9.56	-0.25	-7.16	-5.63	-13.55
541.leela	Og	0.63	-1.61	-2.58	-3.17	-	-	-	-
	O1	-11.26	-11.87	-11.75	-14.10	-9.59	-8.92	-20.49	-25.33
	O2	-1.76	-1.61	-3.46	-4.22	-11.83	-12.23	-12.63	-14.84
	O3	-4.68	-8.65	-9.09	-8.95	-2.61	-13.33	-13.60	-20.60
557.xz	Og	-1.31	-0.66	-2.89	-2.89	-	-	-	-
	O1	-4.26	-3.77	-4.75	-6.80	-1.76	-1.41	-8.21	-9.55
	O2	0.18	0.00	-0.36	0.36	-4.87	-2.84	-3.53	-2.32
	O3	0.36	2.02	2.02	1.09	-1.63	-4.90	-3.89	-10.97

TABLE XIII
DEBUG INFORMATION METRICS FOR THE TEST SUITE (AVG.) AND PERCENTAGE IMPROVEMENT FOR Ox-dy COMPILER CONFIGURATIONS ON SPEC CPU 2017. PARETO-OPTIMAL SOLUTIONS ARE HIGHLIGHTED IN BOLD.

compiler	configuration	Og		O1		O2		O3	
		prod. of metrics	Δ (%)	prod. of metrics	Δ (%)	prod. of metrics	Δ (%)	prod. of metrics	Δ (%)
gcc	Ox	0.6251	-	0.5543	-	0.4191	-	**0.3999**	-
	Ox-d3	**0.6723**	7.56	0.6264	13.01	**0.5434**	29.67	**0.5214**	30.39
	Ox-d5	0.6769	8.29	**0.6331**	14.22	**0.5684**	35.63	**0.5471**	36.81
	Ox-d7	0.6776	8.41	**0.6426**	15.92	0.5769	37.66	0.5476	36.96
	Ox-d9	**0.6785**	8.55	**0.6560**	18.33	**0.5782**	37.96	0.5692	42.35
clang	Ox	-	-	0.5467	-	0.5164	-	**0.5121**	-
	Ox-d3	-	-	0.6013	10.00	0.5830	12.91	**0.5477**	6.95
	Ox-d5	-	-	0.6167	12.80	**0.5920**	14.64	0.5930	15.80
	Ox-d7	-	-	**0.6542**	19.67	**0.5978**	15.76	**0.5947**	16.14
	Ox-d9	-	-	**0.6735**	23.20	**0.6160**	19.29	**0.6288**	22.79

TABLE XIV
SPEEDUP (COMPUTED W.R.T. O0) AND RELATIVE PERFORMANCE LOSS OF Ox-dy COMPILER CONFIGURATIONS ON SPEC CPU 2017. PARETO-OPTIMAL SOLUTIONS ARE HIGHLIGHED IN BOLD.

compiler	configuration	Og		O1		O2		O3	
		speedup	Δ (%)	speedup	Δ (%)	speedup	Δ (%)	speedup	Δ (%)
gcc	Ox	2.0390	-	2.2433	-	2.5316	-	**2.6302**	-
	Ox-d3	**1.9449**	-4.62	2.1811	-2.77	**2.4981**	-1.33	**2.6001**	-1.14
	Ox-d5	1.9249	-5.60	**2.1879**	-2.47	**2.4641**	-2.67	**2.3691**	-9.92
	Ox-d7	1.9232	-5.68	**2.1648**	-3.50	2.2764	-10.08	2.3627	-10.17
	Ox-d9	**1.9256**	-5.56	**2.0402**	-9.05	**2.2821**	-9.86	2.3598	-10.28
clang	Ox	-	-	2.2752	-	2.6036	-	**2.6265**	-
	Ox-d3	-	-	2.1936	-3.59	2.4399	-6.29	**2.6046**	-0.83
	Ox-d5	-	-	2.1674	-4.74	**2.462**1	-5.43	2.4525	-6.62
	Ox-d7	-	-	**1.9980**	-12.18	**2.4491**	-5.93	**2.4528**	-6.61
	Ox-d9	-	-	**1.8511**	-18.64	**2.2506**	-13.56	**2.2283**	-15.16

Additional AutoFDO data: The analysis of the AutoFDO SPEC CPU 2017 experiments given in Section V-C considers for brevity only the most performant O2-dy configurations.

Table XV covers all the O2-dy configurations, detailing the speedups over a standard O2 build of the resulting AutoFDO binaries, and the relative speedup variation for our O2-dy configurations over the AutoFDO-O2 baseline.

As referenced in the discussion from Section V-C, here we

TABLE XV

COMPLETE PERFORMANCE RESULTS FOR SPEC CPU 2017 BINARIES OPTIMIZED WITH AUTOFDO USING O2 AND OUR Ox-dy CONFIGURATIONS FOR PROFILE COLLECTION (SPEEDUP COMPUTED OVER O2 BINARIES). DETAILS ON FURTHER STEPPABLE LINES IN THE O2-dy PROFILING BINARIES.

benchmark	O2	O2-d3			O2-d5			O2-d7			O2-d9		
	speedup	speedup	Δ (%) speedup	Δ (%) steppable extra lines	speedup	Δ (%) speedup	Δ (%) steppable extra lines	speedup	Δ (%) speedup	Δ (%) steppable extra lines	speedup	Δ (%) speedup	Δ (%) steppable extra lines
500.perlbench	1.0832	1.0810	-0.20	5.53	1.0789	-0.40	6.17	1.0962	1.19	6.33	1.0940	0.99	6.73
502.gcc	1.0562	1.0562	0.00	8.93	1.0562	0.00	9.35	1.0562	0.00	9.46	1.0637	0.70	10.12
505.mcf	1.0721	1.1114	3.54	7.72	1.1050	2.97	8.26	1.1001	2.55	9.99	1.0970	2.26	10.79
523.xalancbmk	1.0017	1.0122	1.04	5.19	1.0122	1.04	5.67	1.0070	0.52	5.86	1.0087	0.69	5.91
525.x264	0.7410	0.7483	0.98	4.91	0.7410	0.00	5.44	0.7434	0.33	5.62	0.7434	0.33	6.10
531.deepsjeng	0.9975	0.9975	0.00	13.21	1.0025	0.50	13.34	1.0025	0.50	13.27	1.0025	0.50	14.40
541.leela	1.0123	1.0159	0.35	4.77	1.0141	0.18	5.12	1.0232	1.06	5.39	1.0250	1.23	11.36
557.xz	0.9927	1.0167	2.36	6.51	1.0055	1.27	6.79	1.0074	1.45	6.97	1.0055	1.27	7.23
average	0.9886	0.9988	0.0103	6.6843	0.9956	0.0071	7.1482	0.9982	0.0096	7.4821	0.9986	0.8523	8.6497

show per-benchmark percentage improvements in the number of steppable lines in the O2-dy binaries used for profile collection. This serves as an indicator of the higher amount of source lines covered by debug symbols. The improvements generally grow with the amount of disabled passes, with the sole exceptions of O2-d5 and O2-d7 for 531.deepsjeng. This consistency backs our argument that disabling the passes identified by DEBUGTUNER on the test suite is effectively increasing the quantity of produced debug information also on the SPEC CPU 2017 benchmarks. Nevertheless, the isolated instances of performance loss may advise of potential issues in how the compiler processes the more complete profiles.