Accurate Coverage Metrics for Compiler-Generated Debugging Information

J. Ryan Stinnett
King's College London
London, United Kingdom
jryans@gmail.com

Stephen Kell
King's College London
London, United Kingdom
stephen.kell@kcl.ac.uk

Abstract

Many debugging tools rely on compiler-produced metadata to present a source-language view of program states, such as variable values and source line numbers. While this tends to work for unoptimised programs, current compilers often generate only partial debugging information in optimised programs. Current approaches for measuring the extent of coverage of local variables are based on crude assumptions (for example, assuming variables could cover their whole parent scope) and are not comparable from one compilation to another. In this work, we propose some new metrics, computable by our tools, which could serve as motivation for language implementations to improve debugging quality.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Compilers*; *Correctness*.

Keywords: debug information, optimisation

ACM Reference Format:
J. Ryan Stinnett and Stephen Kell. 2024. Accurate Coverage Metrics for Compiler-Generated Debugging Information. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction (CC '24), March 2–3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3640537.3641578

1 Introduction

Compilers emit debugging metadata, or "debug info", to enable the mapping of machine program states back to source program states. This information is consumed not only by interactive debuggers, but also by profilers, tracers (e.g. SystemTap [9]), coverage tools, etc. Toolchains have developed standard formats for it, such as DWARF [8], which decouple tools from compilers. Unfortunately, *compiler optimisations* interact poorly with debug info: the information produced by production-grade compilers is often incorrect or missing after optimisation [1, 4, 5, 7, 11, 13, 16, 28, 30, 32].

CC '24, March 2–3, 2024, Edinburgh, United Kingdom

When this happens, the tool is unable to identify correctly the source-level program state of interest. In debuggers, a common failure of this kind is the message "variable optimised out" when attempting to print or evaluate a local variable. This message often occurs even when a variable remains represented in the program; it is triggered when the debug info, not the variable, is missing. When coverage is lacking like this, the effectiveness of the tooling is degraded.

Debugging information for optimised code matters. Some codebases cannot be built without optimisation (e.g. the Linux kernel), and others cannot meaningfully be run without it (e.g. resource-heavy programs such as games which are unusable without optimisation). Some bugs occur only on optimised "full speed" code, and some tools are only useful on the same (e.g. profilers). Programmers are used to deploying partial workarounds when facing difficulties debugging optimised code—notably, rebuilding without optimisation—but this brings costs to developers (e.g. rebuilding takes time) and does not help in-the-field bug reporting by end users (being optimised, deployed binaries are frequently undebuggable).

Whereas compiler benchmarks provide a basis for evaluating the optimisation benefit achieved for a program, there is no equivalent way to measure (or show absence of) the incurred *debuggability disbenefit* in the generated "program plus debug info". Certain crude metrics do exist but, as we will survey, they have various flaws and, most notably, are not comparable across compilers. One reason for this is that although missing debug info is usually regarded as a compiler bug, it is currently not clear what it means for debug info to be fully complete and correct. Historically, a "best effort" approach has prevailed, pursuing specific improvements [3] but placing no strong correctness criterion on compilers.

In this paper we develop the first robust approach to measuring the coverage of *local variable information* in DWARF debugging information. Our contributions are the following:

- a model of optimisation under debugging as "residualising" computation, with an analysis of local variables and their life-cycle within a debuggable program (§3);
- a discussion of a series of candidate *local variable coverage* metrics (§4), with experience of applying them to complex cases found in real debug info, culminating in an implemented coverage tool which improves on earlier metrics by obtaining an accurate and achievable "complete coverage" baseline (§5);

- experimental evidence showing that our metric reliably reflects changes in debuggability across compiler versions (§6.1), can explain the debuggability effects of compiler changes (§6.2) and can reproduce *with expected differences* findings of a prior study (§6.3).

Our experiments and implemented tools are available in a deposited artifact [29] (not peer-reviewed).

2 Understanding Debug Coverage

What kinds of coverage could we measure, and what distinguishes coverage from correctness? We discuss these here.

2.1 Distinguishing Coverage from Correctness

Given "full" debug info known to be complete and correct, we would expect it to satisfy the following properties:

Control coverage It is possible to stop at all reachable points in the source program's control flow. Put differently, if code actually executes, then it also appears to execute as observed from the debugger.[1]

Variable coverage At any such point, it is possible to examine all named values that the source program deems to be *in scope* at that point and also taking a well-defined value—roughly, it is not uninitialized. (From here we use "variable" to refer to named values even if the value happens to be immutable, such as a `const` local in C.)

Semantic consistency When examining a variable at such a point, its observed value should be (somehow) consistent with the source program's semantics.

Whereas the first two properties (coverage) are reasonably precise, semantic consistency can be seen as a spectrum of possible correctness conditions: exactly *how consistent* one should expect observations to be might be answered in strong terms ("as if executing the source program unoptimised") or relatively weaker terms, e.g. permitting optimisers to reorder code. (An analogy exists here with memory models, whose consistency properties may be stronger or weaker. Prior work [33] has given names to two polar-opposite properties: "expected" means that even when optimisations are enabled, the debug-time view appears identical to unoptimised execution, while "truthful" means that the debugger shows whatever states are actually inhabited by the optimised program, translating them into source terms as best it can but applying no particular correctness criterion.)

In this work we focus on measuring the *extent* of local variable debug info (coverage) but not its semantic consistency (correctness). Although this means a compiler could game our metric by adding spurious debug info, developers are unlikely to add this intentionally since it would offer an especially bad user experience for end developers.

[1] From here on we assume an imperative source language, i.e. one with explicit control flow, although we believe this could be generalised—perhaps in terms of a partial order on active program constructs.

```
0xb90..0xbb3: (reg RDX)
0xbb3..0xbdb: (value (div (- (reg RAX) (reg RDI)) 4))
0xbdb..0xc0e: (reg RDX)
0xc0e..0xc1a: (frame_offset -24)
```

Figure 1. How DWARF might describe a local variable over four distinct address ranges within a function. Most expressions compute where it is *located*: in a register or (later) on the stack. Over the second range, however, it is not represented explicitly; its *value* is computed as a scaled difference of two registers. (The textual syntax is for illustration only.)

2.2 Debug Info as Residualised Code

Compilers may optimise code so that a variable is no longer explicitly represented. Such variables remain coverable at debug time in modern debug info formats like DWARF [8], which describe them as functions to be computed by the debugger. In DWARF these are *expressions* in an interpreted stack machine language. Expressions can compute a variable's *location* in memory or the register file—perhaps a simple offset from the stack pointer, but sometimes complex (e.g. in a nested function, traversing a static-link pointer to reach locals in the lexically enclosing scope)—or they can compute a variable's *value* directly. Fig. 1 shows an example.

Compilers may also optimise code so that an intermediate control-flow position of the source program is elided, having no corresponding program counter position in the object code. Again, full variable coverage over these positions often remains feasible, as debug info can represent intermediate states to be synthesised by the debugger, effectively existing *in between* the instructions of the object program. Fig. 2 shows an example. This facility is less well established but is implemented in (at least) extensions to DWARF that are proposed for the next standard and already used by GCC [25–27]. Currently, eliminated *branching* control flow cannot be represented in DWARF or any proposed extension.

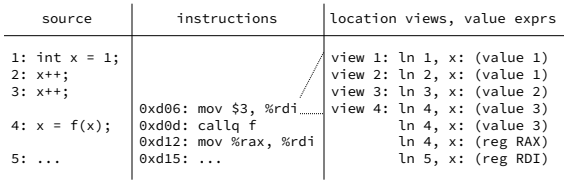


Figure 2. How DWARF can conceptually residualise control-flow positions that were eliminated during optimisation, using the "location views" extension of Oliva [25]. A single program counter value (here `0xd06`) can have a numbered sequence of "views" such that a single local variable is described differently for each view. At debug time, control appears to pass through each view in sequence, even though the program counter does not advance.

Both of these can be thought of as a kind of "residual computation": code eliminated from the program is in effect reinstated in the debug info. We view compilers as outputting two artifacts: the object program proper, and its debug info. Elimination from one does not imply elimination from the other, and in fact the opposite is true: the more thoroughly a variable is "eliminated" from the emitted program, the more it needs to be *described* in the debug info. Any notion of full coverage needs to reflect the potential for residualisation: it is a loss of coverage if the compiler did not take the opportunity to residualise a local variable over some reachable range of positions in the source program. Given adequate residualisation features, optimisations and debugging need not be mutually excluding.

2.3 Research Questions

Using this perspective of debug info *residualising* otherwise eliminated or simplified computation, we can now state the specific research questions to be answered in this paper.

RQ1. For measuring coverage, what is a viable conceptual basis? Clearly coverage can be measured as a fraction of what is *covered* relative to what is *coverable*, but how to determine these is unclear. More specifically: given a program and its debug info, what configurations may a variable v inhabit at a program point p? We will develop a case analysis to answer this question (§3).

RQ2. For measuring coverage, what are the necessary practical steps? How can we measure the extent of local variable coverage in debug info, in a way that fairly represents the effective debug-time availability of the program's local variables? To answer this, we need to define an accurate baseline for when a variable *should* be present, i.e. one that accounts for residualisation opportunities. This is complicated by the sometimes unpredictable and counterintuitive behaviour of compilers and the partial information they generate. We define a measure of coverage with respect to a given local variable, capturing over how much of the program the debugger is able to show a value for it, relative to the achievable baseline. We improve on previous metrics by ensuring the baseline *is* achievable, using both conceptual insights (§4) and experiences during implementation (§5).

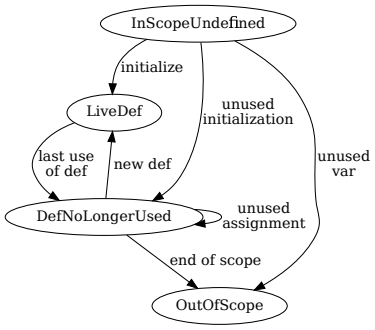


Figure 3. Classical view of the life cycle of a local variable.

RQ3. In aggregate, how does our metric's picture of debuggability depart from those computed by naïve variations and/or previously proposed metrics? We will answer this open question both by applying our metric to both real and synthetic programs and exploring the results in plots, and by a *replication study* in which we use our metrics to reproduce an experiment from recent literature.

RQ4. In detail, does our metric explain debuggability gaps in a way consistent with how these are understood by real compiler developers? We expect low scores in our metric to be indicative of compiler bugs, and higher scores to be indicative of their absence. We study two real fixed compiler bugs before and after their fixes and explore how these are reflected by our metric.

3 A Conceptual Basis for Coverage Metrics

At a high level, measuring coverage means computing the following:

$$coverage = \frac{\#covered}{\#coverable}$$

This demands answers to three questions: what counts as "covered", what counts as "coverable", and in what unit these are counted. In this section we answer the first two of these, by proposing definitions for when a local variable is *covered* and *coverable*, made by analogy with a familiar liveness analysis using the data-flow method.

3.1 Liveness as a State Space

Traditional liveness analysis deems a variable to be live at a point if a *definition* reaches that point and has at least one later *use*. We can model each variable in such a scenario by two unary predicates: whether it has been initialized ("Defined" or D) and whether the current definition will be used again ("Live" or L). Although these two predicates are orthogonal, a reduction applies since conventionally "Live" is assumed to imply "Defined" (i.e. uninitialized reads are not considered), leaving three in-scope states. The resulting machine is shown in Fig. 3; for readability, an additional "out of scope" exit node is added.

3.2 A More Realistic Life Cycle

In a debugging scenario, the situation is more complex in a few ways.

Multiple deaths. As before, a variable may be defined (initialized) or not, but "dead" is a less clear concept. An in-scope variable that is deemed dead by the optimiser may still be requested by the debugger's user, even though the program does not need it. From here we qualify "dead" as "program-dead", to highlight this. Consider the straight-line program shown in Fig. 4. On the right is the source code, and on the left is a flow chart of significant events in the

Table 1. Case analysis of in-scope variables, which at any point may (A) be allocated to a storage location, (D) take a defined value according to source semantics, (L) hold a program-live value, and (K) be possibly knowable given the right debug info.

A	D	K	L	short name	example/notes
0	0	0	0	InScopeOnly	just come into scope; neither allocated nor defined
0	1	0	0	Unknowable	no storage, value not program-live, no longer recoverable from other state
0	1	1	0	KnowablePDead	not program-live nor allocated, but recoverable as a function of other state
0	1	1	1	UnallocatedPLive	normal case of non-allocated live variable
1	0	0	0	AllocatedUninit	uninitialized reg or stack slot
1	1	0	0	AllocatedStale	program-dead store eliminated; storage still allocated
1	1	1	0	AllocatedPDead	program-dead variable but correct value still stored
1	1	1	1	NormalPLive	normal case: program-live, allocated variable

variable's lifecycle, showing its two additional deaths: when it is unrecoverable (unknowable), and when it is no longer in scope.

Allocated vs residual. Unlike a traditional data-flow analysis we care to distinguish whether optimisation passes have residualised a source variable into a debug info expression or simply represented it directly in some allocated storage (a register or stack slot).

Actual versus potential. Variables are routinely residualised into expressions, but imperfect debug info may miss opportunities to do so. It matters whether a source variable *can be* expressed in this way or not. (This links back to "multiple deaths": one of the two "later deaths" is when there is no such function of program state.)

To account for all these distinctions, we can identify the following largely orthogonal predicates of an in-scope variable:

- Is it "allocated" to any storage location (A)? (If not, it might still be computed by an expression.)
- Is it initialized i.e. ever-defined (D), according to source semantics?
- Is its current definition [program-]live (L)?
- Does it have a value that is knowable (K) from the current object program state, i.e. as any expression over it?

These four elements are again somewhat non-orthogonal. To be knowable and to be live both require being defined. (We regard uninitialized values as meaningless and therefore trivially unknowable.) To be allocated and program-live is also to be knowable (simply by loading from the allocated storage; a DWARF "location expression" denotes this). After applying these reductions, eight states of a possible 16 remain; the state space is shown in Tab. 1.

"Knowable" here means that reading from storage and/or evaluating *some* debug info expression *could* obtain the variable's value. This subsumes three sub-cases: the allocated case ("knowable" simply by loading from storage), the "recoverable" case when a variable is computable as a function of other (redundant) state (as the "scaled difference of pointers" in Fig. 1), but also the "literal value" case e.g. in

```
int i = 0; ... i = ...;
```

… where from the initialization, the debug info simply records the literal zero, no storage needs to be allocated, and no "function" of program state *per se* is required.

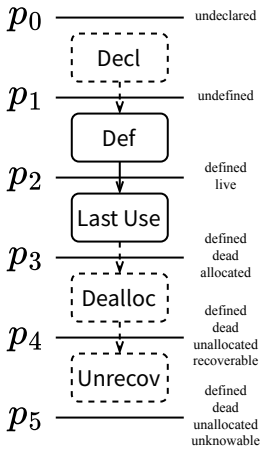
The restriction of K to functions of the *current* object program state reflects how under current tools, when state is to be dropped from the object program, there is no mechanism that can cause it to be remembered e.g. by an attached debugger. Dropped state is not residualised. Whether there exists any such function appears undecidable in general, posing a problem for our metric—which seeks to determine whether each variable is possibly cover*able*. For now, we assume that all local variables are coverable at all points where they are defined, but with optional special handling for the positions after their last use (i.e. when they are finally program-dead). This can avoid penalising compilers for discarding state as a usual register allocator would, with the rationale that this *could* still be covered by a debugger using a technique we describe later (§6.4).

Finally we define two further predicates:

- $S(p)$ is true iff a variable is in scope at p;
- $B(p)$ is true iff the debugging information descri*b*es a variable at p.

Note that B is true when the debug info *actually does* describe the variable, either in its allocation or as a residualised expression—as distinct from whether it potentially could (K). As in Fig. 3, when a variable is out of scope, the other predicates are no longer of interest.

From here, for notational convenience we will identify a predicate with the set of program points where it is true. We can write $B_v(p)$, say, to refer to whether variable v is described at a point p, or we can write simply B_v to refer to the subset of program points $p \in P$ where $B_v(p)$ is true. Here P is the set of all program points as somehow defined; exactly what should constitute a "program point" will be discussed in the next section.




```
int example(int n) {
  // p0: before `x` enters scope (undeclared)
  int x;
  // p1: before live range of `x` (undefined)
  x = 1;
  // p2: within live range of `x` (defined, live)
  int b = x + 2;
  // p3: after live range of `x` (defined, program–dead, allocated)
  ...
  // p4: later, once `x`'s storage is re–used (defined, p–dead...
  f(b); // ... , unallocated, but `x` is recoverable as b – 2)
  // p5: later still, `x` is no longer recoverable...
  // (defined, program–dead, unallocated, unknowable)
  return 0;
}
```

Figure 4. Variable x is in different states across a range of program points p. The right-hand C source function is annotated with these points. The left-hand flow chart illustrates x's lifecycle in a hypothetical compilation, from before x's introduction (Decl), across its definition (Def), last use (Last Use) and the subsequent reclamation of its stack or register storage (Dealloc). A final Unrecov event models the point from which there is no longer enough state to reconstruct the value.

4 Defining a Coverage Metric

In this section we begin with metrics already found in the literature, progressively identifying and eliminating issues with them, culminating in our proposed metric.

4.1 Naïve Instruction-Based Metrics

A naïve but easily computed metric simply counts the number of instruction bytes over which the value is described (B) by debug info, as a fraction of a total possible number of instruction bytes over which it is in scope (S).

$$C_v = \frac{\left| B_v \right|}{\left| S_v \right|} \text{ for } P \text{ the program's set of instruction bytes}$$

This is what is computed by existing tools such as `llvm-dwarfdump` [18] and `debuginfo-quality` [23]. Unfortunately this results in four problems.

Firstly, measurements are **not comparable** across compilers, or even across differently configured runs of the same compiler. Instruction counts reflect details of the compilation, such as how many instructions were used to realise a particular feature of the source program, which vary independently of how debuggable the result is. Essentially this metric places a varying, compilation-specific weighting on the coverage available across parts of the program, according to how many instructions the compiler used to realise them.

Secondly, in the presence of *location views*, as shown in Fig. 2, it simply **omits to count some coverage**. Location views can cover a source variable over a range of zero instructions, which by definition will not be counted.

Thirdly, in practice it **accidentally favours some compilations**, specifically unoptimised compilations which put local variables in a stack slot for their whole lifetime. Conversely it penalises optimised compilations using a register allocator. This is because a variable described as being located in a stack slot will be in B (it is described) over all program points in S, even those where the variable is not yet defined (points not in D). Conversely, a variable that is stored only in registers will be in B only where it is also in D because registers are allocated not over a whole function but over specific ranges of instructions—*live ranges*, which always start at a *definition*, so naturally exclude program points not in D. Put differently, it overestimates coverage by counting as covered a variable in a described stack slot that holds only garbage (uninitialized) data ($B \supset D$). (Usually a "variable not defined" message would be preferable in these cases!) This overestimation means **full coverage becomes impossible to achieve** when using register allocation, since the denominator counts program points over which the variable will have no described value so cannot be counted in the numerator ($B \subset S$). Fig. 5 reveals the unachievable coverage which other tools suggest exists but is ultimately unattainable after accounting for each variable's defined region.

4.2 Counting Source Lines, not Instructions

To address the first two problems—incomparability across compilers, and omitting "source-only" coverage—we can count source lines, not instructions. This applies to both the numerator and the denominator.

$$C_v = \frac{\left| B_v \right|}{\left| S_v \right|} \text{ for } P \text{ the program's set of source lines}$$

The debug info includes a line table, supplying a mapping such that any set of source lines can be projected to a set of instructions or vice-versa. By performing this projection,

we obtain a set of source positions (e.g. pairs of (filename, line number)) that should be embodied by *any* compilation of the same code. When P is defined in this way, the number of instructions embodying any particular source location may vary arbitrarily without affecting the metric. This is intentional: we deem instruction counts not relevant to the experience of a developer debugging at source level.[2]

Line tables' mappings are not one-to-one: inlining leads to source lines mapping to many program counters, while various code-folding transformations do the converse. While we may still calculate sets of source lines or instruction bytes, a local variable may be covered or not covered at each. The simplest scheme (which is what we implement) calculates a variable's coverage as an unweighted mean across all realised instances of a source line, e.g. inlined copies plus the out-of-line copy. Each source line continues to contribute at most 1 to the metric's numerator, so in the case of such "multiply instantiated" lines, a contribution may be fractional rather than just 0 or 1.[3] (One could also argue for unequal weighting among these instances, e.g. supposing the out-of-line copy is called more often than each inlined copy. Making a meaningful choice would require profile information.)

[2]The focus on lines does mean that very long source lines, e.g. as produced by some macro expansions, suffer limited resolution. However, our metric could work with any notion of *program points* that is reflected in the compiler's line table. For example, source features could be correlated to DWARF column information, but compiler support for this information is patchy.

[3]For exposition we continue to use the set cardinality notation, in spite of this generalisation to fractional contributions. Another fractional case might be where only certain bytes of a value are available, as common with locals of `struct` or other composite type. Our tool does not currently account for this, but could easily be extended to do so.

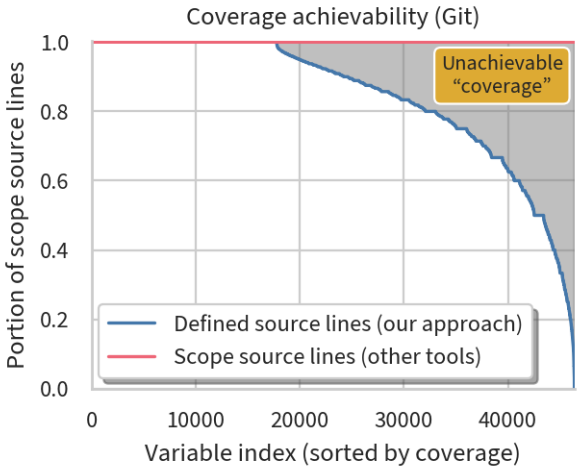


Figure 5. Coverage achievability in the Git codebase. The gap between the lines represents "artificial" coverage that other tools suggest exists but cannot be achieved as it is outside the variable's defined ranges. The "other tools" line is always 1.0 by definition and included only to emphasise the difference between approaches.

4.3 Correcting Accidents, Permitting Full Coverage

To correct the accidental favouring of stack allocation and unachievability of full coverage, the obvious evolution is to apply a "definedness filter" to both numerator and denominator. We call this "scope shrinking". (This can be done while counting either source lines or instruction bytes; we use only the former from here on.) We simply omit to count, in either numerator or denominator, those program positions for which the variable has no defined value.

$$C_v = \frac{\left| B_v \cap D_v \right|}{\left| S_v \cap D_v \right|} \text{ for } P \text{ the program's set of source lines}$$

Unlike the earlier metrics, this is challenging to implement. How should we calculate D_v, the set of source lines in the *defined range* of variable v? A form of binary liveness analysis could be used, but would require control-flow reconstruction on the binary, and would be assuming correctness of the compiler-generated line table.

5 Implementation

This section outlines how our implementation avoids a complex binary analysis and avoids assuming correctness of the compiler's line table, by adopting an external baseline.

Heuristic attempts. To avoid the complexity of control-flow reconstruction the LLVM tool's developers trialled a modification where, essentially, an approximation of $|S \cap D|$ was used. If S consists of a set of instruction bytes, as offsets within a function, the denominator is $|S \setminus [0, n)|$ where byte n is the first over which *any* DWARF expression for the variable is defined. However, this is circular: it uses what is *covered* (the instructions covered by defined DWARF expressions) as a proxy for what is *coverable* (the instructions over which the variable *should* be covered), undermining the intention of a coverage metric to capture the gap between these. Indeed the heuristic was found to be unreliable and removed [2].

Need for an external baseline. To avoid this circularity and answer more authoritatively which source positions *should* be covered, we can use an external reference, in the form of some unoptimised version of the program. This could be either source code or unoptimised intermediate representation. We use this unoptimised version to calculate the sets of source positions S and D needed for the denominator.

`mem2reg` *as a baseline.* To approximate D by excluding those lines which *precede the variable's point(s) of first definition*, one method is to use the compiler itself. Although a fully unoptimised build does not help (because the compiler usually just places all variables on the stack for their entire scope), adding *just* LLVM's `mem2reg` pass to the unoptimised `O0` build largely has the right effect: it moves variables into registers and describes them only from their point(s) of definition. Essentially, this exploits `mem2reg`'s liveness analysis

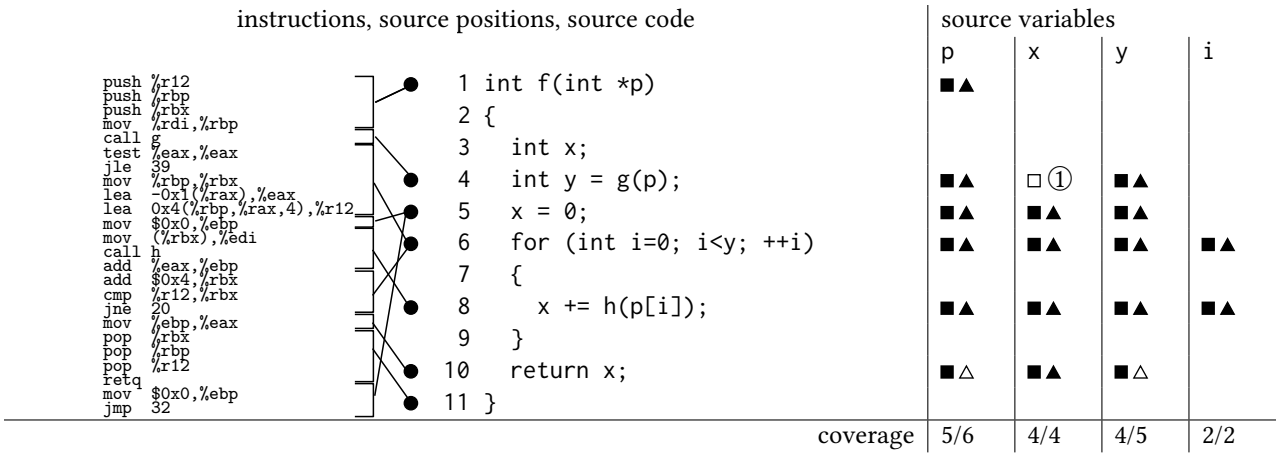


Figure 6. An annotated view of how our coverage metric might view a simple compiled function. Circles ● denote source positions identified in the debug info. Square boxes mark those source positions for which a variable is in-scope and undefined (□) or defined (■). Triangles mark those for which the given source variable is covered by debug info (▲) or not (△); the markers shown are typical for a lightly optimised version of the code. ① marks a location where variable `x` is not yet initialized but, if allocated storage e.g. on the stack, would be erroneously counted as covered by a naive metric.

to approximate $S \cap D$. Since this pass works on virtual registers, it is not limited by the supply of physical registers, so is applied to any non-address-taken scalar local. We call this baseline `O0-mem2reg`. Unfortunately, various uncontrollable effects make `O0-mem2reg` unsuitable as a baseline. At different optimisation levels, we found that the line number tables map the same code to slightly different source ranges and that `O0-mem2reg` tends to yield a slight subset of the true D. Some `O1` compilations report source lines absent from the `O0-mem2reg` baseline set, leading to coverage over 100%, which is clearly neither accurate nor acceptable.

Source analysis. Instead of `mem2reg`, we use a source-level analysis to calculate S and D as sets of source line numbers. These sets are constructed by a series of "filters" over the source line number space. We classify AST nodes according to whether they *perform computation*—for example, declaration-only nodes do not perform computation, but nodes using variables and invoking operators do. (Note that this computation may later be residualised into debug info; there is no obligation for this computation to be embodied by *instructions* in the eventual binary.) We use the parser's retained source coordinates to map nodes to lines. Any *non*-computational lines that appear in the DWARF line table are ignored when calculating the measured binary's set of described lines (B), sidestepping the `O0` and `O1` variability seen with `mem2reg`. The source analysis similarly computes S and D as subsets of the file's computational lines. Since S and D are determined by source analysis, our denominator remains valid even when compilers' line tables are incomplete.

Static versus dynamic analysis. The approach we have described is a static analysis examining all code; it does not rely on collecting execution traces. However, in the presence of unreachable code it may be fairer to filter out lines that are not executed across some collection of test inputs. Our method is adaptable straightforwardly to this approach, simply by an additional filter on the sets of lines in the numerator and denominator. We describe an experiment using this approach in the next section.

6 Evaluation

Of our four research questions, RQ1 and RQ2 have been addressed by construction in the preceding sections. In this section we first outline some practical experience with our metric across a range of codebases and compilers/versions, then consider the remaining questions RQ3 and RQ4.

6.1 Experience with Our Metrics

Fig. 7 shows coverage achieved for the Git codebase as compiled by Clang 15 and GCC 13, as measured using our metric. Coverage is computed per variable and plotted after sorting the variables by coverage. Each compiler is run at multiple optimisation levels. As one might expect, when comparing coverage across optimisation levels, we see that any optimisation beyond `O0` significantly degrades coverage. GCC's `Og` optimisation level (which is a variant of `O1` tuned for better debugging) offers the best coverage of the optimised runs. Clang does not currently offer a meaningful `Og` (it is merely an alias for `O1`), but the LLVM community is working on [17] adding this in the near future. Recent versions of Clang enable almost all optimisations at `O1`, so the higher levels show only minor differences. The difference between `O0` and `O0-mem2reg` (which moves most variables off the stack and into registers) is intriguing, since it shows a sizeable amount

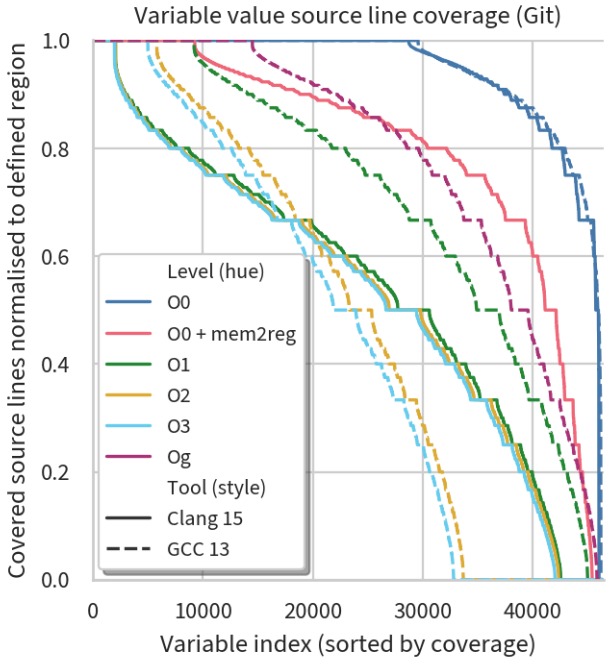


Figure 7. Variable value source line coverage for Git codebase. For each optimisation level, variables are sorted by coverage (variable's index may be different for each line).

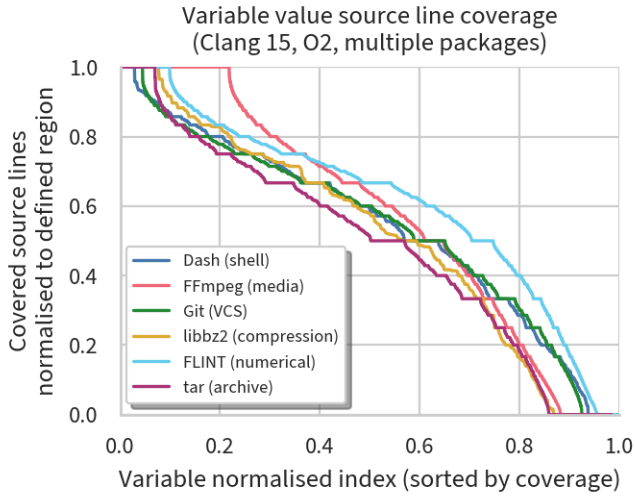


Figure 8. Variable value source line coverage across several packages. Variables are sorted by coverage.

of coverage lost even after little optimisation. GCC fully covers more variables than Clang at the same setting, but its coverage then drops more rapidly.

Fig. 8 shows coverage across 6 analysed packages. These packages were chosen to explore coverage across different application domains. The list includes Dash (shell), FFmpeg (media), Git (VCS), libbz2 (compression), FLINT (numerical), and tar (archive). Compiled with Clang 15 at O2, we see a similar profile across all packages.

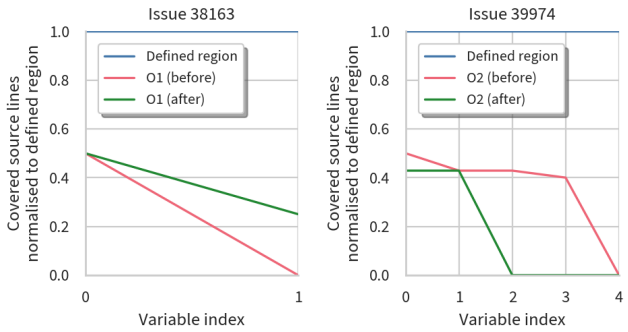


Figure 9. Coverage before and after resolving LLVM issues 38163 (left) and 39974 (right). Each measured program is the (very small) bug reproduction example attached to the corresponding issue, with only 2 and 5 variables respectively.

We compared our coverage metric approach with that of other tools (`llvm-dwarfdump`, `debuginfo-quality`) for the Git codebase using Clang 15 at O2 by computing the correlation between the two. The Pearson correlation coefficient is 0.656, which suggests only a moderately correlated positive relationship. Indeed, our approach often computes quite different coverage values than past approaches [18, 23]. Several factors explain this. As highlighted earlier (§4), we lift from bytes to lines, restrict to computational lines, and include only the defined region as covered and coverable. These improvements affect both numerator and denominator, and manifest differently for each variable depending on whether each line of its defined region is covered or uncovered.

6.2 Case Studies

RQ4 asked: in detail, does our metric explain debuggability gaps in a way consistent with how these are understood by real compiler developers? We examine this through two case studies. Compiler changes can go both ways: even when an issue is *resolved*, coverage may go up or down. This is because compiler authors generally take the (quite reasonable) perspective that incorrect debug info is worse than none at all [19], so may sometimes remove coverage to avoid incorrectness. We study one case of each kind.

Debug info repaired. LLVM issue 38163 ("Loop strength reduction preserves little debug info") involves the `LoopStrengthReduce` pass which transforms loop induction variables into more efficient forms. Debug info describing these induction variables was being dropped. The issue was resolved via a change which residualises the variable in some cases. As seen in Fig. 9, our metric shows that source variable value coverage improved as a result.

Debug info dropped. LLVM issue 39974 ("Salvaged memory loads can observe subsequent memory writes") concerns a function that loads from memory into an unused local variable. The `EarlyCSE` pass eliminates the load but correctly residualises this as a dereference operation in debug info.

However, after a later store to the pointer this expression yields an incorrect value: the newly stored value, not the value that would have been loaded earlier (which has become potentially unknowable). The compiler authors chose to remove the debug info for this variable. Fig. 9 shows that our metric confirms that several source variables lost all value coverage as expected for this compiler change.

6.3 Aggregate Comparisons: A Replication Study

RQ3 asked: in aggregate, how does our metric's picture of debuggability depart from those computed by naïve variations and/or previously proposed metrics? We answer this by a *replication study* in which we adapt our metrics to reproduce an experiment from recent literature. The similarities we find serve as validation of definitional and implementational correctness but we also find certain expected differences that we can explain in terms of our metric's definition. The study reproduces the experiments of Assaiante et al. [1] which generated the data plotted in Fig. 10. The metrics presented have been computed across the same 5,000 Csmith-generated programs from their published artifact. Their metrics examine local variables only, not formal parameters, so we created a similarly adapted version of ours.

While we have defined our own metric in a per-variable fashion, aggregating across lines, Assaiante et al. did the converse, defining a metric per line and aggregating over that line's variables. They first compute "line coverage" as a ratio of the lines present in the DWARF line table for a given compilation relative to the lines present in their `O0` baseline. Next, they compute "availability of variables" by attempting to stop in the debugger on each such line, and recording the number of variables accessible relative to `O0`. The overall availability score is the arithmetic mean. Fortunately, our method is straightforwardly adapted to a line-oriented approach, simply by enumerating at each line the familiar denominator $|S \cap D|$ (the variables that are both in-scope and defined) and numerator $|B \cap D|$ (the variables that are both described and defined). Our approach is able to compute the metric for `O0`, since we use our familiar source analysis as a baseline, whereas Assaiante et al. use `O0` as the baseline; we noted in §5 why this is unreliable.

Their work relies on a dynamic analysis and thus will not encounter unreachable lines, which these generated programs do contain. For this replication experiment, we added a binary analysis step using a simple custom Valgrind [22] tool to enumerate executed lines, as anticipated earlier (§5). This allowed us to obtain similar line coverage.

Similar trends appear in both our metric and theirs when looking across compiler versions and optimisation levels. Our line coverage is slightly lower owing to the different baseline. Since our metric can measure line coverage even at `O0`, we can additionally see that if their `On` coverage is multiplied by the `O0` value (which is their baseline), we indeed arrive at (approximately) our own `On` line coverage. Meanwhile, our availability of variables is somewhat higher. This reflects the expected improvement of our approach: by limiting coverable lines to the defined region, we avoid the problem of the over-counting stack-based `O0` baseline, which artificially prevents reaching 100% coverage (§4.1).

6.4 Experimenting with End-Of-Scope Treatments

Scope shrinking (§4.3) disregards a variable's region of undefinedness at the beginning of its scope, i.e. until it is defined. A near-dual scenario exists at the *end* of its scope: a defined variable may have a period of *unknowability* when its last value, although well-defined in source terms, is no longer recoverable from the object program state (seen in Fig. 4). Since DWARF expressions residualise computation, but not state, even the most sophisticated DWARF expression could not reconstruct the variable in this case. However, a hypothetical debugger could be implemented today that performs this *knowledge extension*, by saving a variable's value at its last moment of knowability (computed from the instruction ranges of debug info). Fig. 11 shows a simulation of the potential coverage gains after applying end-of-scope knowledge extension. We do not know of any such debugger, although DWARF version 5 alludes to such possibilities (in the specification of `DW_OP_entry_value`, the operation to obtain *somehow* a value as it existed on entry to the function) and *omniscient debuggers* (e.g. Pernosco) take the extreme position of ensuring full knowability at all times.

7 Related Work

To our knowledge, relatively little work has addressed the problem of measuring debug info coverage. As covered earlier (§4.1) two existing tools `llvm-dwarfdump` [18] and `debuginfo-quality` [23] compute essentially the same instruction-based metric, whose problems we surveyed. Assaiante et al. [1] used a differential approach which tracks how many variables can be accessed in the debugger on each source line relative to the unoptimised version. Our metric offers more precision via our source analysis baseline and by considering only each variable's defined region.

Rather more work has addressed other aspects of source-level debugging of optimised code. Research into debugging optimised programs made some progress in the 1980s [11], mainly focused on tracking the location of source variable values in optimised program state during execution. DOC [5] stores debug info tracking locations of values from memory through to registers, resembling an early form of today's debug info. Zellweger [33] describes how to compare unoptimised and optimised program data-flow and control-flow to untangle state changes of specific optimisations and recover values. OPTVIEW [30] avoids attempting to map between source and binary, but instead lifts the optimised program back up to a modified source view. Still other systems achieve source-level ("expected") behaviour by temporarily switching to an unoptimised version of function(s) while a debugger

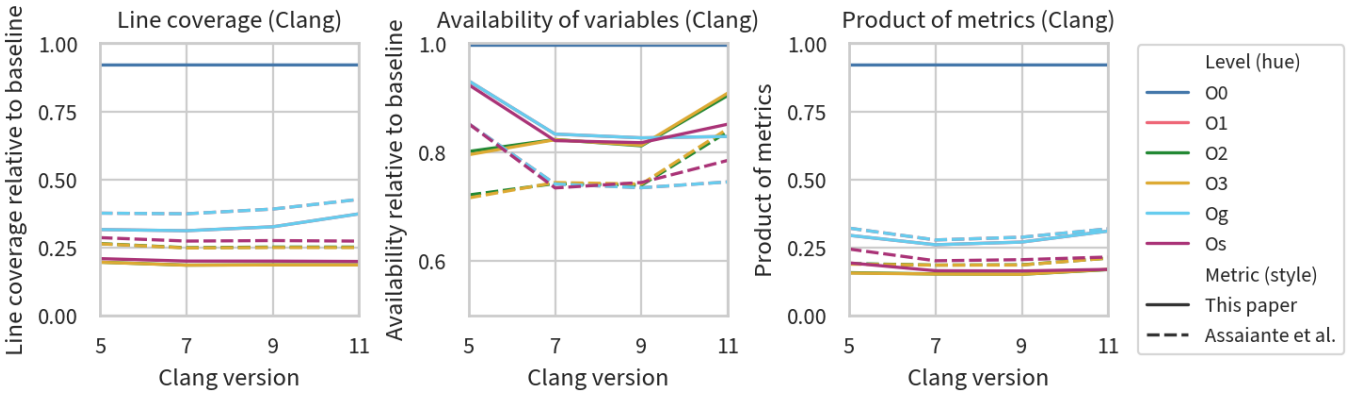


Figure 10. Comparison of our metric approach (using static source analysis as baseline) with Assaiante et al. [1] (using `O0` as baseline). Metrics computed using the same 5,000 Csmith-generated [31] programs as used by Assaiante et al. Of the two, only our metric is capable of measuring coverage at `O0`. The two metrics show the same trends across optimisation levels and versions but a markedly different absolute value, reflecting our fairer "scope shrinking" baseline (§4.3).

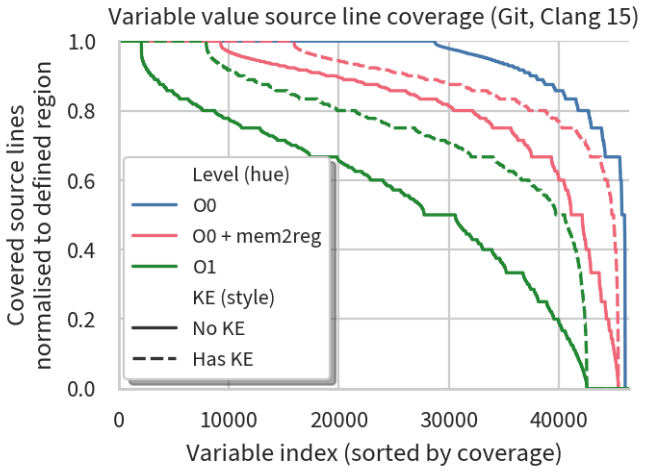


Figure 11. Coverage for the Git codebase before and after end-of-scope "knowledge extension" (KE).

is attached [12, 34], including also performing the switch mid-function using on-stack replacement [6, 10, 12]. These systems sidestep the issues motivating our work, but rely on just-in-time compilers, and also constrain the compiler to retain specific program points where it is safe to jump from optimised to unoptimised mode for debugging. By contrast, ahead-of-time toolchain compilers are traditionally implemented without such constraints, motivating our work.

We also discussed earlier (§6.4) how a debugger could currently perform "knowledge extension" and thereby improve the debugging illusion for any given metadata. Although we know no debugger doing this per se, *omniscient debuggers* such as Pernosco, built on record/replay systems such as `rr` [24], offer an extreme case: knowledge is kept indefinitely. Recording must be enabled at the start of execution, and on current hardware it must also narrow the envelope of executions e.g. by serialization of multithreaded code.

8 Conclusions and Future Work

We have defined various properties of correct debug info, characterising debug info as a "residualisation" of the optimised-away parts of the original program. We have elaborated these properties into coverage criteria that we have embodied in an implemented tool and found useful in finding and explaining real debuggability problems.

The viewpoint of "residualisation" is one we find especially useful. It suggests two specific areas of future work: *designing for state retention* and *designing for residualisation*.

The first is to allow a conventional debugger to "residualise state", i.e. to preserve local variables over ranges when they would otherwise be *unknowable*, as we developed in §6.4. Although today's debug info already makes this possible, doing it reliably would require control-flow reconstruction in the debugger (since the numerically "last" instruction in an address range might not be the last to execute) and ideally a faster breakpoint mechanism (such as that of Kessler [14]), to avoid the slowdown of frequent traps.

The second goal is open-ended, beginning by observing how compilers have not been engineered from the viewpoint of "optimisation as residualisation" but rather "optimisation as elimination". The generation of debug info is an additional manual task for the author of an optimisation pass. Could it instead be an automatic side-effect of the primitive IR transformations provided by a compiler framework? This stands to align debug info generation with verified compilation [15] and translation validation [20, 21].

Acknowledgments

We thank Maxine Champion, Al Grant, Robert O'Callahan, and the anonymous reviewers for their helpful feedback. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) via grant EP/W012308/1.

References

[1] Cristian Assaiante, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. 2023. Where Did My Variable Go? Poking Holes in Incomplete Debug Information. In *Proc. of ASPLOS '23*. https://doi.org/10.1145/3575693.3575720

[2] Kristina Bessonova. 2019. [llvm-dwarfdump][Statistics] Unify Coverage Statistic Computation. https://reviews.llvm.org/D70548

[3] Ronald F. Brender, Jeffrey E. Nelson, and Mark E. Arsenault. 1998. Debugging Optimized Code: Concepts and Implementation on DIGITAL Alpha Systems. *Digital Technical Journal* 10, 1 (1998), 81–99.

[4] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. 1992. A New Approach to Debugging Optimized Code. In *Proc. of PLDI '92*. https://doi.org/10.1145/143095.143108

[5] Deborah S. Coutant, Sue Meloy, and Michelle Ruscetta. 1988. DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code. In *Proc. of PLDI '88*. https://doi.org/10.1145/53990.54003

[6] Daniele Cono D'Elia and Camil Demetrescu. 2018. On-Stack Replacement, Distilled. In *Proc. of PLDI '18*. https://doi.org/10.1145/3192366.3192396

[7] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. 2021. Who's Debugging the Debuggers? Exposing Debug Information Bugs in Optimized Binaries. In *Proc. of ASPLOS '21*. https://doi.org/10.1145/3445814.3446695

[8] DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format: Version 5. https://dwarfstd.org/doc/DWARF5.pdf

[9] Frank Ch. Eigler. 2006. Problem Solving With SystemTap. In *Proceedings of the Linux Symposium* (Ottawa, Canada).

[10] Shu-Yu Guo. 2014. Debugging in the Time of JITs. https://rfrn.org/~shu/2014/05/14/debugging-in-the-time-of-jits.html

[11] John Hennessy. 1982. Symbolic Debugging of Optimized Code. *TOPLAS* 4, 3 (July 1982), 323–344. https://doi.org/10.1145/357172.357173

[12] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proc. of PLDI '92*. https://doi.org/10.1145/143095.143114

[13] Jakub Jelínek. 2010. Improving Debug Info for Optimized Away Parameters. In *GCC Summit*. https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=jelinek.pdf

[14] Peter B. Kessler. 1990. Fast Breakpoints: Design and Implementation. In *Proc. of PLDI '90*. https://doi.org/10.1145/93542.93555

[15] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[16] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. 2020. Debug Information Validation for Optimized Code. In *Proc. of PLDI '20*. https://doi.org/10.1145/3385412.3386020

[17] Stephen Livermore-Tozer. 2023. [RFC] Redefine Og/O1 and Add a New Level of Og. https://discourse.llvm.org/t/rfc-redefine-og-o1-and-add-a-new-level-of-og/72850

[18] LLVM Project. 2020. llvm-dwarfdump - Dump and Verify DWARF Debug Information. https://llvm.org/docs/CommandGuide/llvm-dwarfdump.html

[19] LLVM Project. 2022. Source Level Debugging with LLVM. https://llvm.org/docs/SourceLevelDebugging.html

[20] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proc. of PLDI '21*. https://doi.org/10.1145/3453483.3454030

[21] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proc. of PLDI '00*. https://doi.org/10.1145/349299.349314

[22] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of PLDI '08*. https://doi.org/10.1145/1250734.1250746

[23] Robert O'Callahan. 2018. Comparing the Quality of Debug Information Produced by Clang and GCC. https://robert.ocallahan.org/2018/11/comparing-quality-of-debug-information.html

[24] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *Proc. of USENIX ATC '17*. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[25] Alexandre Oliva. 2010. Consistent Views at Recommended Breakpoints. In *Proc. of GCC Summit*. 6. https://www.fsfla.org/~lxoliva/papers/sfn/gcc2010.pdf

[26] Alexandre Oliva. 2017. Location View Numbering. https://dwarfstd.org/ShowIssue.php?issue=170427.1

[27] Alexandre Oliva. 2017. Statement Frontier Notes and Location Views. https://developers.redhat.com/blog/2017/07/11/statement-frontier-notes-and-location-views

[28] Alexandre Oliva. 2019. GCC gOlogy: Studying the Impact of Optimizations on Debugging. https://www.fsfla.org/~lxoliva/writeups/gOlogy/gOlogy.html

[29] J. Ryan Stinnett and Stephen Kell. 2024. Accurate Coverage Metrics for Compiler-Generated Debugging Information (artifact). https://doi.org/10.5281/zenodo.10568392

[30] Caroline Tice and Susan L. Graham. 1998. OPTVIEW: A New Approach for Examining Optimized Code. In *Proc. of PASTE '98*. https://doi.org/10.1145/277631.277636

[31] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proc. of PLDI '11*. https://doi.org/10.1145/1993498.1993532

[32] Polle T. Zellweger. 1983. An Interactive High-Level Debugger for Control-Flow Optimized Programs. In *Proc. of SIGSOFT '83*. https://doi.org/10.1145/1006147.1006183

[33] Polle T. Zellweger. 1984. *Interactive Source-Level Debugging of Optimized Programs*. Ph. D. Dissertation. University of California, Berkeley. https://search.library.berkeley.edu/permalink/01UCS_BER/1thfj9n/alma991002570669706532

[34] Lawrence W Zurawski and Ralph E Johnson. 1991. Debugging Optimized Code with Expected Behavior. Unpublished draft.

Received 2023-11-13; accepted 2023-12-23