Debugging Debugging Information using Dynamic Call Trees

J. RYAN STINNETT, King's College London, United Kingdom
STEPHEN KELL, King's College London, United Kingdom

Debugging tools rely on compiler-generated metadata to present a source-language view, but current compilers often throw away or corrupt debugging information in optimised programs. Attempts to test debugging information are confounded by ad-hoc limitations of the debug info formats and a lack of clarity on whether the compiler or the format is to blame for any given loss. Adopting the "residual program" conceptual view of debug info, we conduct a study of the quality of debugging information in respect of the *source-level dynamic call trees* it can recover. We compare the trees recovered from optimised and unoptimised versions of the same program, producing a classification of the observed divergences. For each class, we analyse whether format or compiler is to blame and identify specific ways to address these defects. We also validate our classification across a larger collection of well-known codebases.

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

Additional Key Words and Phrases: debug information, optimisation

ACM Reference Format:
J. Ryan Stinnett and Stephen Kell. 2026. Debugging Debugging Information using Dynamic Call Trees. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 105 (April 2026), 27 pages. https://doi.org/10.1145/3798213

1 Introduction

Developers understand and improve programs with the help of *source-level dynamic analysis tools*, including debuggers. Such tools provide the illusion of observing a running *source* program, even when what runs is really a compiled binary program for the target machine. Toolchains for C, C++ and other languages have decoupled compilers from tools by evolving standard metadata or "debug info" formats, such as DWARF [2017], which map machine states back to source program states.

Unfortunately, debug info is often degraded or removed by optimisation passes in modern compilers [Assaiante et al. 2023; Brooks et al. 1992; Coutant et al. 1988; Di Luna et al. 2021; Hennessy 1982; Jelínek 2010; Li et al. 2020; Oliva 2019; Stinnett and Kell 2024; Tice and Graham 1998; Zellweger 1983]. In these cases, debugging tools easily diverge from showing a complete and source-consistent view of execution. A popular workaround is to debug only an unoptimised build, but there are many scenarios where this is not feasible: optimised version of a program: in-situ debugging or profiling of a production system, resource-heavy programs such as games that are unusable without optimisation, codebases cannot be compiled without optimisation (e.g. Linux kernel, GNU C library), or where the bug only surfaces after optimisation.

All interactive debuggers, and some profiling and tracing tools, provide a view based in part on the *dynamic call tree* [Ammons et al. 1997] of an execution. For example, "stepping into" or "stepping out of" a call refer respectively to continuing until the current branch of this tree is extended downwards or until execution returns to the parent node. Breakpoints set "on a function"

Authors' Contact Information: J. Ryan Stinnett, King's College London, London, United Kingdom; Stephen Kell, King's College London, London, United Kingdom.

ACM 2475-1421/2026/4-ART105
https://doi.org/10.1145/3798213

```
void receive_for_stream(SyncQueue *sq,
    unsigned int stream_idx, SyncQueueFrame frame) {
  SyncQueueStream *st = &sq->streams[stream_idx];
  if (st->samples_queued >= frame_samples(sq, frame))
    av_fifo_drain2(st->fifo, 1);
  st->samples_queued -= frame_samples(sq, frame);
}
int frame_samples(
    const SyncQueue *sq, SyncQueueFrame frame) {
  return (sq->type == SYNC_QUEUE_PACKETS)
      ? 0 : frame.f->nb_samples;
}
```

```
  submit_encode_frame
    sq_receive
      receive_internal
        receive_for_stream
          frame_samples
          av_fifo_drain2
-         frame_samples
    encode_frame
      update_benchmark
```

Fig. 1. In this program snippet extracted from FFmpeg, we see a function `frame_samples` that is called twice in `receive_for_stream`. When optimisation is enabled, these two calls may be reduced to one by common subexpression elimination (CSE). Optimisations such as this cause the dynamic call tree observed in the debugger to differ from the source program's and from the programmer's expectations.

fire at the moment the tree gains a particular new branch, while a "backtrace" is a path up to the root of the tree. Programmers' mental abstraction of the system correlates heavily to this tree structure—or at least, to the *source-level* version of this tree. The binary-level version of the tree may differ, both superficially (e.g. by name mangling of the node labels) but also *structurally*, following optimisations such as inlining or tail calling. Modern debugging tools undertake to recover a source-level view from the binary-level execution: they not only demangle names but also show inlined and tail calls, using features of the debug info that represent these. Over 30+ years, the DWARF format has steadily accreted new features for handling various optimisations of these kinds.

Unfortunately, these efforts currently stop well short of recovering the expected source-level view. Developers are familiar with various problems when debugging at higher optimisation levels: breakpoints not being hit, failure to "step out" of an inlined call, stepping out too far, single calls appearing to be entered multiple times or vice-versa, or sometimes even an entirely different function being called than that written in the source program.

These divergences pose a dilemma: since there is no clear expectation about which might be compiler bugs and which are simply "expected" or "the best a debugger can do" in the face of arbitrary optimisations, compiler developers have no established correctness criterion for whether their optimisation correctly transforms debugging information. This paper explores how *dynamic call trees* may be used to define a strong correctness criterion. Our contributions are as follows.

- We conduct an exploratory study, by constructing a dynamic analysis tool that can collect source-level dynamic call trees of real executions. We compare the trees collected across optimised and unoptimised executions of various real codebases, and use this to show how modern compiler optimisations rule out simple correctness criteria for the debug info, such as equality with a reference (source-level) call tree.
- We identify over a dozen distinct categories of recurring divergence between the trees seen in optimised and unoptimised executions. Among these, we isolate those that are "non-reportable" or *expected*, in the sense that issues lie in *either* the debug info format *or* (more subtly) compilers' established practices for how that format is used. For each such category

we identify how compilers could resolve the divergence using extended or (in some cases) existing forms of debug info.
- We similarly classify the unexpected divergences, with examples exhibited by our tool, and elaborate their causes in compilers, including bugs we have reported in LLVM and GCC.
- We apply these findings in a further, broader study, measuring the extent and breakdown of our classified divergences across various codebases, compilers, compiler versions, and optimisation levels. This data reveals certain patterns and trends across different compilers, while confirming that our categorisation explains most divergences seen.

2 Background

2.1 Dynamic Call Trees

We assume a source language having a notion of calls and returns. A call begins an activation of a procedure (or "function") and a return ends it. For simplicity of exposition, we will focus on single-threaded sequential execution, although of course the testing we enable also benefits the (sequential) debuggability of multithreaded programs.

A dynamic call tree, as described by Ammons et al. [1997], is a complete history of the calls and returns in a program execution, arranged with calling frames as parents of their callee frames.

2.2 Object- vs. Source-Level Call Trees

An object-level dynamic call tree is induced by machine instructions such as (on Intel architectures) `call` and `ret`. This object-level tree is commonly the one used e.g. in the context-sensitive profiling techniques that originated the idea. (We will continue to use Intel mnemonics, but of course our approach also works on non-Intel architectures.)

However, *source-level* debugging must recover a source-level tree from an object-level execution. The source-level tree additionally includes calls that *did not* appear as `call` and `ret` instructions, but were instead realised by compiler optimisations such as inlining, tail calls, compiler-builtin handling of standard library functions, and so on.

Some existing tools can generate dynamic call trees, including `uftrace` and `funtrace` for user programs, or `ftrace` in the Linux kernel.[1] However, they do not attempt to reconstruct the full source-level tree, nor to work on arbitrary optimised code. Rather, they rely on edge instrumentation (`-pg` or `-finstrument-functions`) which limits subsequent optimisation, instrument only explicit call and return instructions, and will betray optimisations that do occur e.g. in cases of call elimination, substitutions, inlining, and various other scenarios which we survey later (§3).

In contrast to these tools, source-level debuggers such as `gdb` or `lldb` *do* attempt to elaborate a fully source-level call tree via their stepping and breakpointing operations. Source-level debugging relies on debugging information's capacity to "undo" the effect of such optimisations, for example presenting the *illusion* that an inlined call really is occurring much like its out-of-line counterpart.

In order to perform this feat, debugging information is complex. For example, it records which instructions are attributable to each inlined instance of a call. Each will have specific information describing the stack or register locations of local variables, any nested inlined calls, and so on.

2.3 Residualisation and Debug Info for Optimised Code

Modern debug info formats have remarkably sophisticated features for "undoing" the visibility of optimisations. Debugging information is a Turing-powerful computational artifact in its own right: a latent pattern during optimising compilation is that as state or behaviour is eliminated from the

[1]At the time of writing these tools are available and documented: https://uftrace.github.io, https://github.com/yosefk/funtrace, https://www.kernel.org/doc/html/latest/trace/ftrace.html

object program, it "moves into" the debugging information. For example, if an initial zeroing of a variable is eliminated because the zero value is never read, the zero value is recorded in debug info so that the debugger can still print the variable's source-level value. Where computation is transformed (e.g. performing strength reduction to eliminate a loop index variable), expressions in the DWARF can restore the variable even when it is completely absent from the object program, by reconstructing it from the state that does remain (e.g. the "index pointer" that replaces the index variable). This view of debug info as "residualisation" is explained with further examples in earlier work [Kell and Stinnett 2024].

Debug info formats offer not only residual computation but also some residual state. For example, the execution of code that has been entirely optimised out at object level—e.g. not merely an inlined call, but a call with no attributable instructions remaining, perhaps following constant folding or common subexpression elimination—can be resynthesised as different source-level "location views" at a single program counter [Oliva 2017]. In essence, this feature allows the control state of the object code to be re-augmented with additional debug-time program counter positions, each showing different local variable values, source line number, etc.

2.4 Limited Differential Testability of Debug Info

Despite these powerful features, studies confirm that current optimisers cause severe degradation to debuggability. Assaiante et al. [2023] measured local variables' availability at on average 0.15–0.25 of the theoretical maximum, across optimised -O2 and -O3 builds with recent Clang and GCC. We found with the same compilers that the median variable had availability in the range 0.50–0.67 relative to a more realistically achievable baseline [Stinnett and Kell 2024].

It is known that the collection of available debug info features does not amount to a remedy that can "undo" the observed effects of all optimisations. For example, location views can restore eliminated straight-line control flow, but not branching control flow. However, these recent studies also show that compilers do not correctly and maximally use those features that are available, meaning many real bugs *are* fixable within existing debug info formats. (Such fixes are easily found in the bug trackers of open-source compilers.)

Research literature to date has attempted differential testing of debug info only for weak properties. For example, the following property was recently used [Di Luna et al. 2021] for testing backtrace and other dynamic call tree-related features at debug time.

> The Backtrace Invariant is violated when the trace from optimized code contains a step over a line l for which the stack backtrace includes a function name that is not present in the stack backtrace of any step in the trace from the unoptimized code that refers to the same line l.

Although useful for bug-finding, this remains a weak property. To satisfy this invariant, it suffices that the backtrace seen at a source line l always includes only function names that were seen in *some* backtrace arising at line l in the unoptimised program. This means it is satisfied by various incorrect backtraces such as (among others):

- any past or future backtrace reaching line l, even when entirely unrelated to the true backtrace at the given program state;
- a backtrace whose order is randomised or reversed, relative to the true backtrace at that state;
- a backtrace that is empty.

This invariant also requires no relationship between the debugger's "backtrace", printed at some point, and any dynamic trace of calls and returns it has shown previously leading to that point, e.g. by a sequence of "step into" or "finish" operations.

2.5 Compiler Bugs vs. Format Bugs

In this paper we make a contribution towards enabling *strong* differential testability of debugging information. Our insight is that the apparent restriction to weak properties arises from a latent assumption that debug info is a computationally weak artifact: a simple stateless mapping of object program state to the source level. However, this is inaccurate in modern DWARF, whose features have already outstripped this, and whose conceptual extension, under the "residual program" framing of debug info, allows full recovery of a faithful source-level view. This is possible because, *in extremis* a residual program might resort to encoding an entire separate copy of the unoptimised program text, consuming the same input but otherwise maintaining its own state and independently executing an unoptimised variant of the program. (This extreme is unlikely ever to be required in practice, in that even under the heaviest compiler optimisations, at least *some* of the object program's computation and state would remain useful in reconstructing a source-level view.[2]) This existence proof escapes from the horns of the dilemma mentioned in the Introduction: an observed defect is never "unavoidable" in the sense of fundamentally impossible to resolve. Rather, it might reflect a defect in the debug info format itself rather than the compiler, i.e. requiring kinds of residual state or computation that current DWARF is incidentally unable to express.

For differential testing to yield actionable results for compiler authors, some method is needed for distinguishing "format bugs" from "compiler bugs". For example, although it is obviously possible to take the dynamic call tree generated by optimised code and compare it with that from an unoptimized build (whose debug info tends to be far more reliable), when the two trees diverge it must be possible to classify, at least approximately, the reported issues so as to blame either the debug info format (for not making expressible the necessary residual computation or state) or the compiler (for not expressing it, in spite of its expressibility). This paper is, we believe, the first work to satisfy that goal: it provides tools and methods for comparing dynamic call trees in a way that distinguishes compiler bugs from format bugs.

3 Exploring Divergent Call Trees

In this section we consider the following questions. What divergences appear in the dynamic call trees generated by optimised and unoptimised compilations? Among those found, which can be ascribed to the compiler and which to the debug info format? Our answers are gathered from an exploratory experiment over a selection of real codebases and compilers.

3.1 Collecting a Call Tree: The Instruction-wise Algorithm

Conceptually, we construct a dynamic call tree roughly by stepping through the program one instruction at a time, and applying rules that determine whether we have entered or left a call. DWARF debugging information divides the object program text into *subprograms* that occupy known ranges of instructions which we call the subprogram's "instruction space".[3]

Basic algorithm. Out-of-line calls are recognised fairly easily: these are branch instructions that:

- move from one subprogram's instruction space to another's; or
- move within one subprogram's instruction space either by a call (i.e. context-pushing) instruction or by a jump (non-context-pushing) instruction that has been annotated in the debugging information as performing a tail call.

[2]This "recompute separately" approach, already partially adopted by modern debug info, clearly comes at a cost of no longer observing the object program proper. Rather, the debugger is showing an reconstructed "illusion" of source-level execution. To allow observing miscompilation bugs and the like, it should of course remain an option for the debugger's user to observe only the bona fide object-level execution.

[3]A subprogram need not occupy a single contiguous "range" of addresses, hence our preference for "space".

The complexity of the latter case is necessary because a heavily optimised function might internally jump back to its entry point, yet such jumps may or may not be self-calls; we rely on the debugging information to identify these, which luckily DWARF can express.[4]

Out-of-line returns always occur as a return instruction. Our algorithm builds up the tree of calls and returns as they occur.

Inlined calls. When a subprogram's instruction space includes the result of inlining at one or more of the source-level call sites it contains, descriptions of the inlined calls nest *under* the DWARF subprogram record. In effect, at each instruction, there is a nest of zero or more inlined call sites that are active there; we term this the "inlined call chain fragment". Of course, inlined calls need not begin or end with any branch instruction. Our instruction-wise algorithm therefore must ask, at each control flow edge, whether any inlined call(s) have been entered or exited across this edge, and update the tree accordingly. As we will see, current DWARF can only provide an approximation of the information that would be needed to represent all source-level call trees faithfully.

Relationship with frame information. The algorithm we have described so far collects a call tree as execution proceeds, and so can always enumerate a stack trace simply by walking up the collected tree. Since debuggers do not customarily collect call tree history, debug info includes additional *frame information* to enable the debugger to generate a stack trace at any point by walking the active object-level calls up the stack, using saved register values to recover the object-level calling context. The frame information can be seen as a redundant partial representation of the dynamic call tree. However, it does not attempt to recover the *source-level* call chain (e.g. it does not duplicate information about the inlined call chain fragment). We therefore do not attempt to cross-check our call trees against frame information, although prior work has considered the testing and synthesis of frame information separately [Bastian et al. 2019].

Decorating the tree with coordinates. At a minimum, the tree must record the identity of the functions being called. It can also be decorated with other information. We will assume that each call edge should be decorated with the coordinates of the call site, i.e. the call's source file/line/column information, and similarly for the *call target* (e.g. the position of the jumped-to entry point, or the first instruction in the inlined instruction space).

Return information. Although a dynamic call tree proper does not contain back-edges from callee to caller (it would no longer be a tree), building the tree correctly requires recognising returns. When a callee completes we also record the corresponding coordinates of the *return site*, i.e. the location of the responsible `return` statement or equivalent. Dually, there is the *return target* i.e. the location returned to, but this is implied by the call site, so is not collected separately. Therefore our tree records *three* facts for a single completed call: its call site (which we will notate as `CF` for "call from"), its entry point (`CT` for "call to") and its return site (`RF` for "return from"). When a tail call is traversed, as annotated in the DWARF, it is remembered on a stack of active tail calls; the next explicit return generates one return-from entry for *all* such active tail calls.

3.2 Collecting a Call Tree: Implementation

We built tools to collect a dynamic call tree of executions of user-level Linux binaries using the Quarkslab QBDI dynamic binary instrumentation framework [Hubain and Tessier 2017]. This is

[4]A converse issue occurs: some context-pushing call instructions are actually not semantically calls. For example, on Intel, `call 0; pop rX` is the idiomatic way to materialise the program counter into register `rX`. Such instructions therefore require the converse annotation, i.e. to mark a context-pushing instruction as nevertheless *not* a source-level call. Such an annotation does *not* exist in current DWARF. Luckily this idiom is an assembly programmer's trick and our experiments have not encountered it in any compiler-generated code. If they had, it would be featured in Tab. 2.

able to preserve the original instruction boundaries, and in our experience is faster than single-stepping using `ptrace()` by a factor of at least 1000. When testing an early version of this binary instrumentation approach on the Git codebase, we found that our QBDI-based tools ran at ~270x native execution, while a `ptrace()` approach was far slower at ~1,700,000x native execution.

The tracing and instrumentation logic are injected using `LD_PRELOAD`, and operate in pre- and post-instruction hooks. These hooks track the source-level function name, file, line, and column associated with the current instruction, using debugging information loaded as needed. The tree is collected as a trace of call-tree events at instruction granularity.

On calls/jumps to a new function (as defined in §3.1), we push our internal activation record and print "call from" and "call to" events in the trace. On a return, we pop an internal activation record and print a "return from" event in the trace. Inlining may cause multiple stack changes at each instruction; we difference the debug info's "inlined call chain fragment" at the two instructions.

Calls to functions in external libraries outside the program also add complexity. In our experiment we choose to observe a single executable binary at a time, since this is the granularity at which we have been controlling compilation. Therefore, we do not collect our traces within calls to shared libraries. Our tool must handle specially the platform-specific mechanisms, such as procedure linkage tables, used to reach external code.

3.3 Exploratory Study Outline

What do we find when comparing call trees across unoptimised and optimised executions? The method of our exploratory study was as follows.

(1) Collect two call tree traces from different program versions, unoptimised (`-O0`) and optimised (`-O1` or `-O2`). We largely assume that the unoptimised build will reproduce the source-level call tree, but will note some exceptions to this rule shortly (§3.4.1).
(2) Diff the two call tree traces (explained below) using the algorithm of Chawathe et al. [1996].
(3) For each tree edit operation produced by diffing, diagnose either manually or automatically the (usually) one or (sometimes) more discrete divergences that it exhibits. Intuitively, one discrete "divergence" is a manifestation of a *single* compiler behaviour. If manual diagnosis is required, for any such behaviour that we expect to recur, record a tree edit pattern that captures it. Future iterations then automatically diagnose recurrences of the same pattern. Divergences are reported only once per unique source location. (Note that each *pattern* may still be reported many times—once for each source location where it is witnessed.)

This hybrid manual/automated approach allows gradually expanding the search to find distinct compiler behaviours. The collected behaviours are detailed in Tab. 1 and Tab. 2.

Following experience, in order to allow for optimized runs where inlining is enabled, a further preprocessing step was introduced prior to the diffing step. This prevents inlining artifacts from dominating the reported divergences, and is described more fully shortly (§3.6).

Trace entries and patterns. In more detail, each trace entry is a triple:

$$< \text{CT} \mid \text{CF} \mid \text{RF}, \texttt{function}, \texttt{coordinates} >$$

for example

$$< \text{CT}, \texttt{main}, \texttt{hello.c:3} >$$

Each completed call generates three entries (not counting from those for any callees it may have) which we arrange in a tree-structured trace such that call-to and return events are children of the "call from" that activates the call. Each tree edit consists of one operation (add, remove, replace, move) along with the tree item indices needed to apply the operation. Applying all of the tree edits to the "before" tree would transform it into the "after" tree. *Patterns* are predicates over tree edit operations and the related call tree events identified by their indices, capturing properties such as "only coordinates changed" or "deleted CT, CF and RF to the same f, where f is external". The repertoire of predicates such as "external" is limited only by what is represented in DWARF about function f.

Optionally, when running with LLVM, our tools can also make use of LLVM *optimisation remarks* to identify which optimisation pass is responsible for a given divergence, as an additional decoration on tree events. We modified the LLVM-based toolchains used in our experiments to add additional optimisation remarks for cases of interest, such as library calls removed, and these may be used to match certain patterns.

Executions explored. Our exploratory study examined traces from executions of Git, FFmpeg, and tar compiled with Clang 13 and GCC 11. (Later experiments will also examine data from other codebases and more recent compilers—see §4.2.)

Expected and unexpected divergences. Tab. 1 and Tab. 2 show the divergences resulting from our exploratory study. One immediate finding is that there are divergences of two kinds, hence the two tables. We separate *unexpected* divergences, i.e. those that appear to indicate a straightforward compiler bug which could be reported and fixed, from *expected* divergences which result a limitation *either* of DWARF (usually) *or* (in a few cases) of how it is customarily used. Expected divergences cannot be fixed without a change to one of these. For example, a C compiler may turn a call from `printf` to a (faster) call to `puts`, if only one argument is given. There is no immediately obvious DWARF feature for recording these substitutions, and so customarily, nothing is done and it is "expected" (by compiler authors; not necessarily by users!) that a breakpoint set on `printf` will simply not fire where this transformation has been applied. We will itemise a series of enhancements to DWARF that could address these expected classes of divergence.

Separation criteria. To separate "expected" from unexpected divergences, some judgement is required. Some recurring classes ask obvious questions of the compiler: *why doesn't it* generate DWARF of a particular form which, although unconventional, would arguably be a viable representation of the post-optimisation mapping to source level? In such cases, the arguably bug lies not in current DWARF features but in how these are customarily interpreted by compiler authors. For example, to continue the same example, one could view the `puts` call as a specialised *body* of `printf` that has been substituted, a.k.a. inlined, by the compiler. Indeed wrapping a DWARF `inlined_subroutine` record around the call site to `puts` would allow a breakpoint on `printf` to fire at this position. On the other hand, the resulting call stack (`puts` within `printf`) need not model any true source-level call stack. Such widened interpretation of DWARF's `inlined_subroutine` would therefore require discussion among compiler developers; we leave this as an "expected" divergence.

Tree diffing. While a textual diffing algorithm can compare traces line-by-line, these easily generate differences that do not capture the change as valid edits to a call tree. The example in Fig. 2 shows two adjacent calls from `all_attrs_init` in the unoptimised program, one to `container_of_or_null_offset` and another to `hashmap_iter_next`. In the optimised trace, the call to `container_of_or_null_offset` has been optimised away. As shown on the left, a text diffing algorithm may (and indeed we observed this happen in practice) mark the end of one call and the start of another as removed in this case, even though that is not a valid tree edit operation and does not match the change actually performed by the optimiser. By using a tree diffing algorithm in our tools [Chawathe et al. 1996], we see the diff shown in the right side of the figure which selects a single sub-tree as removed, matching the call removed by the optimiser.

Table 1. Categories of trace divergence that are *expected* i.e. arise from optimisations that current Dwarf (or standard practices/interpretations) does not suffice to recover, as identified in our exploratory experiment.

Category	Semantic preconditions	Path to resolution
Coordinates changed by small delta ($\leq$ 3 lines) (§3.4.4)	Various	Locations views and/or dependent coordinates (see §3.4.4, §3.4.5)
Coordinates removed owing to control flow merging (§3.4.5)	Repeated steps across multiple control flow paths	Dependent coordinates (see §3.4.5)
Added library call (§3.4.1) (e.g. array initialisation loop $\longrightarrow$ added `memset` call)	Varies by library call (e.g. aggregate initialised to zero, or other known byte value)	Call could be marked artificial in debug info for skipping
Removed library call (e.g. call to `memcpy` $\longrightarrow$ converted to store)	Specialised transformations for known library functions	Call could be described in debug info like inlining
Replaced library call (e.g. simple `printf` $\longrightarrow$ converted to `puts`)	Specialised transformations for known library functions	Call could be described as original call like inlining (§3.3)
Out-of-line call removed by constant propagation (§3.4.2)	`static` callee (local linkage), callee body available, provably compile-time-constant result	Call could be described in debug info like inlining
Merged multiple out-of-line calls to the same callee (§3.4.3)	User `const`/`pure` annotation *or* callee body available and provably effect-free (read-only memory effects)	Obviated call(s) could be described in debug info like inlining (even if reduced to zero instructions; see §3.4.2)
Spurious entries/exits of a single inlined callee (§3.6)	Callee body available	Inlining metadata could describe entries/exits along control-flow edges; see §3.6
Merged multiple inlined calls to the same callee (§3.6)	Multiple successive calls to the same callee, callee body available	Obviated call(s) could be described in debug info even if reduced to zero instructions; see §3.4.2
Code motion of one non-inlined call site past another (§3.4.6)	User `const`/`pure` annotation *or* callee body available and provably effect-free	Possibly "reorder buffer"-like residual state to delay observation of hoisted call §3.4.6

Table 2. Categories of trace divergence that are *unexpected* i.e. result from compiler bugs. Compilers can resolve these by adding or correcting debug info metadata they emit, using only current formats/practices.

Category	Known to occur when	Cause of bug
Coordinates changed by large delta (> 3 lines) (§3.5.1)	'Single return' or other exit path transformation	Lack of convention for coordinates of function exit
Tail call missing (§3.5.2)	Various (for calls in tail position)	Coding omission or limitation of compiler architecture
Inlined call missing	Various (callee body available, $\geq$1 identifiable instruction remaining)	Inlined call should be described; see §3.6
Coordinates removed despite absence of control flow merging	Various	Coding omission or limitation of compiler architecture

```
  CF: all_attrs_init
-   CT: container_of_or_null_offset
-   RF: container_of_or_null_offset
- CF: all_attrs_init
    CT: hashmap_iter_next
    RF: hashmap_iter_next
```

```
- CF: all_attrs_init
-   CT: container_of_or_null_offset
-   RF: container_of_or_null_offset
  CF: all_attrs_init
    CT: hashmap_iter_next
    RF: hashmap_iter_next
```

Fig. 2. This shows the diff operations produced by different diffing algorithms from the same call tree trace snippets extracted from Git. The result on the left was produced by a typical text diffing algorithm, but the suggested edit violates call tree semantics. On the right, we have the result produced by the tree diffing approach used in our tools [Chawathe et al. 1996].

Case studies. The following sections elaborate on a selection of these in greater detail. In each case we review a small code snippet and excerpt from the trace of call tree events, where `CF` means "call from" (on reaching a call site, with its coordinates), and `CT` means "call to" (on reaching a function entry point, with its coordinates), and `RF` means "return from" (about to exit a function). In some cases, coordinates are omitted for readability.

Inlining. Inlining is handled using a special extra processing step, discussed later (§3.6).

3.4 Expected Divergences

3.4.1 Added Library Call. In Fig. 3, we have a loop that initialises an array element by element with a constant value. The optimiser detects this pattern and replaces it with a call to the library function `memset_pattern16`. Synthesised calls such as this (which are not part of the user-written source) should be marked as such in debug info so that they can be (optionally) hidden from the debugging experience. An extension to debug info formats is needed to achieve this.[5]

We also found a related-but-opposite scenario, which is shown in Fig. 4. We have a function that initialises a struct. When optimisation is disabled, Clang achieves this by synthesising a call to a standard library function (`memset`) that was not part of the user's program. When optimisation is enabled, the initialisation instead occurs inline in the generated code. To our trace comparison tools this appears as a call being removed.

3.4.2 Out-of-Line Call Removed by Constant Propagation. In Fig. 5, `has_dos_drive_prefix` is a function that returns the constant value `0`. Compiler analysis may determine that the call has no side effects and that it returns a statically known value.[6] Subsequently, interprocedural constant propagation may use this fact to replace the call with its constant return value.

Note that the call is never inlined per se. The removed call is not currently described by debug info, meaning at debug time there would be no way to stop in the called function. However, to a user of the code there is no difference this two-step approach to elimination—first an analysis on a function's input/output behaviour, enabling later the replacement of a call by a constant value—and obtaining the same code instead by expanding the callee inline and then simplifying. Put differently, replacing the call by a constant value can be seen as a specialised form of inlining. However, neither GCC nor Clang emits DWARF marking it as such.

[5]This could also be encoded by an unorthodox use of existing information, e.g. by a call site marked as `artificial`—whereas current DWARF reserves `artificial` for "objects and types".
[6]Compilers may differ, but in LLVM, this is known as one kind of "memory effect" analysis, and is computed as part of the `GlobalsModRef` alias analysis pass.

```
void ff_cbrt_tableinit(void) {
  static double cbrt_tab_dbl[1 << 13];
  for (int i = 1; i < 1<<13; i++)
    cbrt_tab_dbl[i] = 1;
}
```

```
  CT: ff_cbrt_tableinit
+ CF: ff_cbrt_tableinit
+   CT: Jump to ext. code (_memset_pattern16)
+   CF: Jump to ext. code (_memset_pattern16)
+     CT: External code
+   RF: Jump to ext. code (_memset_pattern16)
  RF: ff_cbrt_tableinit
```

Fig. 3. Call to external code (`memset_pattern16`) added during array initialisation. The optimiser recognises the manual loop initialisation and replaces it with this library call. This artificial call is not annotated as such (for potential skipping), so a debugger would step into `memset_pattern16` even though it doesn't appear in the original program.

```
void read_very_early_config() {
  struct config_options opts = {0};
  opts.respect_includes = 1;
  opts.ignore_repo = 1;
}
```

```
  CT: read_very_early_config
- CF: read_very_early_config
-   CT: Jump to ext. code (_memset)
-   CF: Jump to ext. code (_memset)
-     CT: External code
-   RF: Jump to ext. code (_memset)
  RF: read_very_early_config
```

Fig. 4. Call to external code (`memset`) added during struct initialisation. When optimisation is enabled, the call is removed and the semantics are handled directly in the same function, so its looks like a removal to our tools. This artificial call is not annotated as such (for potential skipping), so a debugger would step into `memset` even though it doesn't appear in the original program.

Interestingly, this means of eliminating calls is surprising even to compiler developers. An LLVM developer familiar with these optimisations wrote: "it's quite unexpected and (to me) really annoying that SCCP [Sparse Conditional Constant Propagation] will perform this kind of propagation".

Enhancement 1. Use of DWARF `inlined_subroutine` records should not be limited to applications of inlining per se, but to any occurrence where an *out-of-line* call is dropped despite control flow still passing through the call.

Zero-length inlines. One argument against the foregoing enhancement is that there might no longer be any instruction attributable to the call. Consider an inlined no-op call, or a call that can be constant-folded into surrounding code. In such cases, the span of entering and leaving the call might reduce to no instructions at all. However, DWARF still permits an inlined call to be specified as covering no instructions, meaning it can still be used for breakpointing. GCC sometimes generates zero-length inlines, but in our experience Clang does not. A zero-length inline would never appear in a backtrace (in the absence of location views[7]) and would not show up in our instruction-wise algorithm, because it is never active at any instruction. However, they are clearly necessary to model this case, e.g. to allow breakpointing, and would naturally regain stepping when combined with location views.

[7]GCC's willingness to generate zero-length inlines is likely related to its support for location views, although these are a logically separate feature.

```
static int is_dir_sep(int c) {
  return c == '/';
}

static int has_dos_drive_prefix(const char *path) {
  return 0;
}

int is_absolute_path(const char *path) {
  return is_dir_sep(path[0]) || has_dos_drive_prefix(path);
}
```

```
  CT: is_absolute_path
  CF: is_absolute_path
    CT: is_dir_sep
    RF: is_dir_sep
- CF: is_absolute_path
-   CT: has_dos_drive_prefix
-   RF: has_dos_drive_prefix
  RF: is_absolute_path
```

Fig. 5. Call to `has_dos_drive_prefix` removed by constant propagation. The callee is `static` (local linkage) and function body is available in the compilation for analysis, so the compiler deduces this call site can be replaced with the callee's constant return value and removes the call. This is not described in current debug info, so a debugger wouldn't stop in `has_dos_drive_prefix` with the optimised compilation.

Enhancement 2. Zero-length inlined callees should be represented in DWARF, even in the absence of location views.

Location views and inlining. In recent DWARF, even *zero-length* ranges of instructions can be recorded as visiting multiple source-level locations, using the *location views* extension. This may be seen as each instruction optionally annotated with a sequence of "fractional program counters", one per view of the instruction. This can be used for stepping; the "step state" is kept in the debugger, and variables may take different values at each step because their DWARF expressions are indexed by "current view". In principle, each of these "view"-level transitions may have a separate effect on the call tree, including entering and leaving any number of inlined calls.

Enhancement 3. Location views should be adopted, to restore stepping in the case of zero-length inlined callees (among other scenarios).

3.4.3 Out-of-Line Call to Multiply-Called Callee Obviated by User Annotation. In Fig. 6, the function `frame_samples` is called multiple times. It has the user-supplied `pure` attribute, telling the compiler it has no observable side effects. This enables the multiple calls to be reduced down to a single call. This optimisation breaks the debugging illusion, as the tooling would not be able to stop multiple times in the called function as suggested by the program source. Similar to previous examples, Enhancements 1 and 2 would allow these dropped calls to be represented up to breakpointability, and Enhancement 3 could allow some of their internal state to be reconstructed (at least arguments and return values, subject as usual to their reconstructibility from available state).

3.4.4 Coordinates Changed. Small coordinate changes are expected under current compiler practices, as there are various situations where coordinates may change a little with little harm done (e.g. consider how loop strength reduction may move indexing calculations). To avoid being overwhelmed by small changes we define a separate pattern for changes of fewer than 3 lines. Larger changes are deemed unexpected; we see later an example bug that this reveals (§3.5.1).

3.4.5 Coordinates Removed by Control Flow Simplification. In Fig. 7, we see that source coordinates at a call site were removed while merging control paths. There are two conditional branches which each call `gettimeofday_nanos`. Clang's `SimplifyCFG` pass combines the call portion of those

```
__attribute__((pure)) int frame_samples(
    const SyncQueue *sq, SyncQueueFrame frame);

void receive_for_stream(SyncQueue *sq,
    unsigned int stream_idx, SyncQueueFrame frame) {
  SyncQueueStream *st = &sq->streams[stream_idx];
  if (st->samples_queued >= frame_samples(sq, frame))
    av_fifo_drain2(st->fifo, 1);
  st->samples_queued -= frame_samples(sq, frame);
}
```

```
  CT: receive_for_stream
  CF: receive_for_stream
    CT: frame_samples
    RF: frame_samples
  CF: receive_for_stream
    CT: av_fifo_drain2
    RF: av_fifo_drain2
- CF: receive_for_stream
-   CT: frame_samples
-   RF: frame_samples
  RF: receive_for_stream
```

Fig. 6. Similar example to Fig. 1, but modified to use the `pure` attribute. This allows the compiler to eliminate a call even if the callee function body is not in the same compilation unit, i.e. is not available for analysis. The compiler instead trusts the user's `pure`-annotated declaration. As before, the effect enables CSE to remove the second call to `frame_samples`.

```
1   uint64_t getnanotime(void) {
2     static uint64_t offset;
3     if (offset > 1) {
4       return offset + highres_nanos();
5     } else if (offset == 1) {
6       return gettimeofday_nanos();
7     } else {
8       uint64_t now = gettimeofday_nanos();
9       uint64_t highres = highres_nanos();
10      offset = highres ? (now − highres) : 1;
11      return now;
12    }
13  }
```

```
  CT: getnanotime at trace.c:1
- CF: getnanotime at trace.c:8
+ CF: getnanotime at trace.c:0
    CT: gettimeofday_nanos at time.c:1
    RF: gettimeofday_nanos at time.c:4
  RF: getnanotime at trace.c:13
```

Fig. 7. In this example, control flow simplification merges the two calls to `gettimeofday_nanos`, formerly on separate branches, into one call on a single branch. As part of this transformation, the call site source coordinates are deliberately dropped since current Dwarf can represent source coordinates as a function of only the program counter. (This is signified by "line 0"; bona fide line numbers start at 1.)

branches into a single call site. Current Dwarf computes coordinates as a function of the program counter only; they cannot depend on other state that might disambiguate which control flow path had been taken, such as (here) the value of `offset`. A "dependent coordinates" extension to the Dwarf line table could resolve this.

Enhancement 4. Source coordinates should be computable as a function of arbitrary program state, not only the program counter.

3.4.6 Code Motion Affecting the Call Tree. Code motion has long been a problem at debug time, because it causes control flow to "jump around" relative to the source program order—not only visiting lines in a changed order, but often visiting (parts of) the same line many times. This reduces the usability of line-wise stepping.

```
1  int xopen(const char *path, int oflag) {
2    for (;;) {
3      int fd = open(path, oflag);
4      if (fd >= 0) return fd;
5      if (errno == EINTR) continue;
6      exit(128);
7    }
8  }
```

```
  CT: xopen at wrapper.c:1
- RF: xopen at wrapper.c:4
+ RF: xopen at wrapper.c:8
```

Fig. 8. In this example, the coordinates for the early return (line 4) are lost when optimisation is enabled in Clang. The compiler frontend creates a new return path when optimisation is used, and this has a side effect of "sliding" the return coordinates to the end of the function.

By focusing on the dynamic call tree, we sidestep many of these issues because most intraprocedural code motion will not affect the tree. Only when one call site is moved past another does the call tree differ. For out-of-line calls, as with dropping or merging calls seen earlier, this is only sound for a textually known or semantically annotated callee, since otherwise the I/O behaviour of the program might change. For inlined calls, we have already argued that in the near term, these should be summarised in the tree *without* ordering.

A longer-term approach to recovering the source-level call tree would involve *residualised control state*. Rather like how speculative out-of-order execution in CPU cores uses a reorder buffer [Smith and Pleszkun 1988], to keep the necessary state for erasing the evidence of speculation or reordering, which are "running ahead" of the committed program-order execution, so in earlier work [Kell and Stinnett 2024] we hypothesised a stateful debugger that could keep additional state to defer the debug-time evidence of code moved ahead of other code. Although there is no prospect of DWARF providing this in the near future, we note that it is a natural generalisation of location views, extending them to "illusory" *control* state.

Enhancement 5. Generalise location views to allow deferring the apparent effects of hoisted code, i.e. restoring an illusion of source execution order.

3.5 Unexpected Divergences

We now take a look at a few specific examples of compiler bugs changing the dynamic call tree and impairing the debugging experience via unexpected divergences. This category can be resolved by changing compiler architecture as needed to fix these bugs.

Each example shows a small code snippet and excerpt of a dynamic call tree trace. Refer to the earlier case study section (§3.3) for more detail on the syntax used.

3.5.1 Large Coordinate Changes. It is not expected for source coordinates to undergo large changes, which we define as larger than three lines. In Fig. 8, we have a return statement on line 4. Without optimisation, this generates the expected call tree, whereas with optimisation, Clang creates a new return path that uses the end of the function, i.e. its closing brace, as the return coordinates, instead of the coordinates of the return statement. This issue is tracked upstream at the time of writing.[8]

In C-like languages that support returning from the middle of a procedure, there is an ambiguity about which source coordinates should be associated with the return site: those of the closing

[8]https://github.com/llvm/llvm-project/issues/57567

```c
char *xdg_config_home_for(
    const char *subdir, const char *filename) {
  const char *home;
  home = getenv("HOME");
  if (home)
    return mkpathdup("%s/.config/%s/%s",
        home, subdir, filename);
  return NULL;
}
```

```diff
- CF: xdg_config_home_for
+ CF: No info for this address
  CT: mkpathdup
  RF: mkpathdup
- RF: xdg_config_home_for
+ RF: No info for this address
```

Fig. 9. In this example, GCC 11 at optimisation level O2 transforms the `mkpathdup` call above into a tail call. It emits only a partial call site record (the call site PC is missing) to describe this tail call in debug info, which prevents a debugger from showing a virtual tail call frame in its backtrace view, and also confuses our trace collector's understanding of the current frame list as seen here.

brace (end of scope) or of the return statement? This affects what we should expect as the source coordinate of the "return from" entry in our trace.

At its root, the issue is that compilers lack a convention (either collectively or individually) on return coordinates for early returns—does the return occur at the syntactic end-of-block point or the return statement? This convention should be invariant under optimisation.

At first glance, our focus on the call tree may appear to inflate the significance of this issue. As long as control passes *first* over the return statement and then over the closing brace, it may appear secondary whether it next does or doesn't pass over the closing brace. However, there are at least two reasons why this matters. Firstly, consider a programmer wishing to set a breakpoint on "any return path" from a given function. There is no clear expectation about whether this should work, nor consistency on whether it does. Secondly, this transformation is still associated with bugs: for example, in the above example under Clang O1, control is *not* seen to pass over the return statement at all. Stepping through `xopen` starting from the inner `open` call, the debugger moves to the `if (fd >= 0)` on the next line, then jumps to the function's closing brace. A breakpoint on the return statement does not fire. In other words, this coordinate change is associated with a loss of debuggability, albeit one which is not fully explored by considering the call tree alone.

3.5.2 Tail Call Missing. Compilers use Dwarf features to describe tail calls, including details of the tail call site PC and function called. This metadata allows debuggers to show a heuristic backtrace showing activations of tail calls. Unlike inlined call chain fragments (§3.1), the fragment of the call chain made up of tail call activations cannot be recovered from the program counter alone, because the number of activations is unbounded (e.g. in a tail-recursive algorithm; by contrast, a recursive call may only be inlined to a bounded depth). These Dwarf features also allowed us to trace tail calls (and their returns) in our instruction-wise algorithm (§3.1).

In Fig. 9, we see an example function where GCC 11 failed to describe a tail call. The debugging user therefore would not see a (virtual) tail call frame in the backtrace. This missing tail call info also appears as a divergence in our traces. Given complete and accurate info from the compiler, our traces do include these artificial tail call frames.

3.6 Handling Traces with Inlining

Inlined calls present particular problems, because in current Dwarf it is not possible in general to recover how many times a given callee was entered or exited. Consider a case where a single

```c
int global;
void effect(void) { ++global; }
inline void do_it(void) { effect(); }
int main(void) {
  int ctr = 0;
  do { do_it(); ++ctr; } while (ctr <= 1);
  return global;
}
```

Fig. 10. A simple C program where, following routine optimisation, a twice-called inline function is only hit once in the debugger, owing to the instruction-based description of inlines in Dwarf and the corresponding lack of information on control-flow edges entering/leaving inlined calls.

call site of function `B` is inlined into its caller `A`, and over six-instruction span of execution, the active function according to the debugging information is `ABBABA`. Was function `B` called a second time (or more!), or was a single call split during instruction scheduling, say? Either of these could be true; all we really know is that some call of `B` was activated one or more times. Although we can expect multiple inlined call sites to generate multiple `inlined_subroutine` records, which should tell us how many call sites are traversed, multiple activations of a single inlined call site might have been merged, e.g. if the call to `B` was in a loop that was unrolled. Therefore there is no upper bound on how many dynamic calls of `B` might have occurred in the source program to give rise to this six-instruction trace; we have only a lower bound of 1.[9]

It is easy to reproduce this problem with a simple C program such as that of Fig. 10, where both GCC and Clang `O2` will place the two `effect()` calls next to each other, both originating from the same inlined call site, so only a single inline call to `do_it()` can be observed in the debugger.

This leads to a major class of expected divergence: spurious entries and exits of inlined callees. The root of the problem is that using current Dwarf, the debugger cannot distinguish a control-flow transition that *enters* a fresh call to an inlined callee from one that *resuming* a previously entered inlined activation which, following to code motion, has become interleaved with unrelated instructions from the inlining context. Dually, it cannot distinguish *returning* from *suspending* an inlined call. These distinctions could, however, be tracked by the compiler and encoded in the debug info, labelling as enter/resume and exit/suspend the relevant instruction-level transitions (or "instruction-or-view", with location views).

Enhancement 6. Recording in Dwarf which control-flow edges encode entries and exits of an inlined function, as an ordered list per edge. (This could be encoded as pairs of PCs, or pairs of *<PC, view> pairs* when using location views.)

This would allow a single control-flow transition to exit one inlined call and enter a second call, handling the loop-unrolling and `ABBABA` cases just considered.

In the absence of such an extension, we require a way to prevent our tool's reports from being overwhelmed by these spurious inlined-call entries, while also generating reports that are useful as possible in finding *unexpected* inlining-related issues. In short, we do so by preprocessing traces (both optimised and unoptimised) to cluster optimiser-inlined calls together. The intuition of the approach is illustrated in Fig. 11. Trace (i) was extracted from an unoptimised Git execution, while trace (ii) came from an optimised run (including inlining) applying only our usual instruction-wise

[9]This limitation arises in addition to the inability of `inlined_subroutine` records to address location views, identified earlier (§3.4.2).

algorithm, and therefore containing spurious entries. The I-prefixed trace events indicate calls and returns derived from the inlined chain recorded in debug info.

The primary difference between these two traces is spurious entries for `strbuf_avail` and `strbuf_grow`. The naïve inlined trace appears to show additional calls to these, not present in the unoptimised view, but in reality inlining has merely allowed later code motion to interleave the instructions of the calls. In general, our naïve instruction-wise method will report *resumes* as spurious *calls* and *suspends* as spurious *returns*. Given later code motion, inlining can disrupt both the *number* of apparent calls in the tree and the *order* in which they are reported to occur.

This limitation of debug info already requires workarounds in debuggers. For example, when setting a breakpoint on an inlined function, so as not to overwhelm the user with "too many hits", debuggers employ heuristics—typically breakpointing the lowest-addressed instruction associated with each matching inlined call, rather than every control flow transition. Given the nature of optimisations, there is no guarantee that this will result in the expected number of hits. (Whereas our naïve method tends to overapproximate, debuggers' tends to underapproximate.)

These observations suggest that a sequence of three transformations to our trace can collectively hide the vagaries of inlining—at a cost of dropping information about *how many times* and *in what order* the affected call sites were activated.

Clustering Firstly we cluster together inlined call sites' subtrees, so that given an inlined call site `X`, *all* activations of `X` are positioned successively, i.e. later activations are hoisted up, past any intervening calls, to a position following immediately the first seen activation. This erases any interleaving with other inlined calls, but preserves the first call's position.

Merging We may then merge the subtrees for these same-site calls, so that a single call-to node exists for any given call site within the enclosing activation. This erases any spurious multiple entries/exits (but may obscure loop structure in the traces).

Floating Finally we may reposition inlined calls: rather than interleaved with other calls, floated calls (and their subtrees) appear in a canonical order (e.g. alphabetical) directly under their caller's entry-point (call-to) node. This erases other code motion, including interleaving across non-inlined calls.

The third and fourth columns of Fig. 11 show the progression of clustering and then merging. Clustering "unscatters" the colours previously scattered around by inlining. After also merging, the trace is equivalent to the unoptimised original. (Floating is not necessary in this example, but it would address cases e.g. where an inlined call was hoisted earlier relative to another.)

More precisely, to perform clustering as in column (iii), we iterate through the call tree breadth-first looking for any calls within a given caller that have the same call edge (matching call from and to events). The children of any additional calls are moved into a cluster with the children of the first call. In the figure, several calls to `strbuf_avail` and `strbuf_grow` end up clustered together. Near the end, there is a further call to `strbuf_avail`, but this has different "call-from" coordinates, so it is correctly kept separate. Meanwhile, for the merging shown in column (iv), we simply remove the additional call records and concatenate their subtrees under the first subtree for that call site. The merged approach can incidentally erase information about loops, since calls within a loop may be reduced to a single call.

Crucially, in general the same transformation is applied also to the unoptimised (reference) trace, *for those call sites which, in the optimised trace, were subject to inlining*. After this, both traces have been abstracted so as to no longer reveal *in what order relative to other inlines* (clustering), *how many times* (merging) or *in what order relative to all calls* (floating) these particular call sites were activated. Since the unoptimised trace now matches the optimised one, up to expected inlining effects, no diff is reported when these expected inlining effects are the only observable divergences.

```
CT: strbuf_vaddf at strbuf.c:390:0
CF: strbuf_vaddf at strbuf.c:394:7
 CT: strbuf_avail at strbuf.h:140:0
 RF: strbuf_avail at strbuf.h:141:2
CF: strbuf_vaddf at strbuf.c:395:3
 CT: strbuf_grow at strbuf.c:92:0
 CF: strbuf_grow at strbuf.c:99:2
  CT: xrealloc at wrapper.c:127:0
  CF: xrealloc at wrapper.c:135:2
   CT: memory_limit_check at wrapper.c:18:0
   RF: memory_limit_check at wrapper.c:35:1
  RF: xrealloc at wrapper.c:140:1
 RF: strbuf_grow at strbuf.c:102:1
CF: strbuf_vaddf at strbuf.c:401:12
 CT: strbuf_avail at strbuf.h:140:0
 RF: strbuf_avail at strbuf.h:141:2
RF: strbuf_vaddf at strbuf.c:408:1
```

(i) unoptimised

```
CT: strbuf_vaddf at strbuf.c:390:0
ICF: strbuf_vaddf at strbuf.c:394:7
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
ICF: strbuf_vaddf at strbuf.c:395:3
 ICT: strbuf_grow at strbuf.c:91:0
 IRF: strbuf_grow at strbuf.c:0:0
ICF: strbuf_vaddf at strbuf.c:394:7
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
ICF: strbuf_vaddf at strbuf.c:395:3
 ICT: strbuf_grow at strbuf.c:91:0
 CF: strbuf_grow at strbuf.c:99:2
  CT: xrealloc at wrapper.c:127:0
  ICF: xrealloc at wrapper.c:135:2
   ICT: memory_limit_check at wrapper.c:17:0
   IRF: memory_limit_check at wrapper.c:0:0
  RF: xrealloc at wrapper.c:140:1
 IRF: strbuf_grow at strbuf.c:0:0
ICF: strbuf_vaddf at strbuf.c:394:7
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
ICF: strbuf_vaddf at strbuf.c:395:3
 ICT: strbuf_grow at strbuf.c:91:0
 IRF: strbuf_grow at strbuf.c:0:0
ICF: strbuf_vaddf at strbuf.c:401:12
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
RF: strbuf_vaddf at strbuf.c:408:1
```

(ii) naïve inlining

```
CT: strbuf_vaddf at strbuf.c:390:0
ICF: strbuf_vaddf at strbuf.c:394:7
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
ICF: strbuf_vaddf at strbuf.c:395:3
 ICT: strbuf_grow at strbuf.c:91:0
 IRF: strbuf_grow at strbuf.c:0:0
 ICT: strbuf_grow at strbuf.c:91:0
 CF: strbuf_grow at strbuf.c:99:2
  CT: xrealloc at wrapper.c:127:0
  ICF: xrealloc at wrapper.c:135:2
   ICT: memory_limit_check at wrapper.c:17:0
   IRF: memory_limit_check at wrapper.c:0:0
  RF: xrealloc at wrapper.c:140:1
 IRF: strbuf_grow at strbuf.c:0:0
 ICT: strbuf_grow at strbuf.c:91:0
 IRF: strbuf_grow at strbuf.c:0:0
ICF: strbuf_vaddf at strbuf.c:401:12
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
RF: strbuf_vaddf at strbuf.c:408:1
```

(iii) with clustering

```
CT: strbuf_vaddf at strbuf.c:390:0
ICF: strbuf_vaddf at strbuf.c:394:7
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
ICF: strbuf_vaddf at strbuf.c:395:3
 ICT: strbuf_grow at strbuf.c:91:0
 CF: strbuf_grow at strbuf.c:99:2
  CT: xrealloc at wrapper.c:127:0
  ICF: xrealloc at wrapper.c:135:2
   ICT: memory_limit_check at wrapper.c:17:0
   IRF: memory_limit_check at wrapper.c:0:0
  RF: xrealloc at wrapper.c:140:1
 IRF: strbuf_grow at strbuf.c:0:0
ICF: strbuf_vaddf at strbuf.c:401:12
 ICT: strbuf_avail at strbuf.h:139:0
 IRF: strbuf_avail at strbuf.h:0:0
RF: strbuf_vaddf at strbuf.c:408:1
```

(iv) also with merging,
matching unoptimised

Fig. 11. An example of how a naive approach to inlining overwhelms the optimised trace with unhelpful expected differences. Distinct colours label each distinct outgoing call edge in the source-level execution. Trace (i) is unoptimised, while trace (ii) is optimised and includes inlining. The `I`-prefixed events denote inlined chain call and return events. After inlining (ii), this code appears to make four calls to `strbuf_avail` and three to `strbuf_grow`. Only two and one respectively are true calls; the others simply mark where each's instructions have been interleaved after inlining. The duplicated call-edge events for each true call are given a distinct colour, to highlight the interleaving. The following two columns show this trace after our inlining preprocessing modes. Trace (iii) shows the "clustering" step, which groups multiple calls along the same call edge together. Trace (iv) shows the addition of the "merging" step, which removes repeated calls. This recovers the original trace; a further "floating" possible step is described in the main text but not needed in this example.

Conversely, given an *unexpected* divergence within an inlined call (e.g. consider an out-of-line callee that was tail-call-optimised but not described as such in the debug info), this would still be revealed by diffing the postprocessed traces (e.g. the affected out-of-line calls would appear in the unoptimised trace, underneath the clustered/merged inlined callee nodes, but would be missing from the equivalent place in the optimised one).

The transformations are also applied transitively. For example, consider a single inlined call site calling `f` that is executed twice, e.g. in a loop within the same parent activation, and each time makes a further inlined call to `g` from the same site. The merged result would feature a single instance of a call to `f` and, under that, a single instance of a call to `g`. Merging all these is necessary because the debug info does not describe, for example, whether the loop was unrolled, whether common subexpressions were eliminated across the different calls, etc. The only expected-reliable facts are that within the containing out-of-line activation, one or more calls to `f` and `g` occurred, from the given inlined call sites. Again, if the debug info could label particular control-flow edges as entering or leaving inlined calls, count and (some) ordering information could be restored.

The desire to filter away the expected artifacts of inlining must be balanced against any true bugs that might be hidden by doing so. Based on experience, in our tool we opted to apply clustering but not merging or floating. The reason is that we already have a pattern-based mechanism for suppressing reports about "expected" issues, and these patterns are sufficient to handle duplicated and reordered calls. Clustering, by contrast, helps recover locality which assists matching these patterns. The compromise is necessary because there remain certain *unexpected* issues that manifest as spuriously repeated or reordered calls. One of the compiler bugs we found in our later experiments (see §4) is of this kind (LLVM issue 156700): an inlined callee that is called only once, but contains a loop, was appearing to be multiply-called after inlining. This was owing to problems with the debug info around the loop header. Under merging, we would be erasing the extra calls that revealed this bug, making it no longer detectable. In our subsequent experiments, we apply clustering only.

3.7 Limitations of the Study Method

Nondeterminism. Sources of nondeterminism (multithreading, signals, time, etc.) can lead to divergences between traces which are not related to the compiler optimisations we analyse here. Nondeterminism here includes control flow sensitive to memory placement, since this will inevitably differ across optimised and unoptimised builds, whose code has different sizes.

The experimental results presented here avoid multithreaded nondeterminism by forcing single-threaded execution. However, our approach can be used to analyse execution from multiple threads naturally and straightforwardly by generating a separate trace for each thread assuming thread identifiers are available that allow comparison between the two executions under analysis.

Use of `-O0` compilation as a reference. We saw in §3.4.1 that compilers may insert "artificial" calls to library functions, such as `memset` or `memcpy`. We correctly detected divergences here, but rather fortuitously, in the sense that it was, unusually, the optimised code that was "more correct" and reproduced the call tree of the source program. Our differential method is dependent on the unoptimised code providing a good reference to source semantics. For example, if a compiler synthesises such calls *and* preserves them across optimisation, we would not detect any divergence.

4 Quantifying Divergence Across Compilers

Using the pattern base created during our exploratory study, we now conduct a larger experiment answering two questions. Firstly, to what extent do the issues identified in our exploratory study explain the divergences seen across a wider selection of compilers, codebases, and the call trees they generate? Secondly, how do the number and mix of observed divergences vary across codebases and across distinct compilers and their versions?

We explored call trees obtained from executions of test suites from several larger codebases (FFmpeg, Git, SQLite) and an established benchmark suite (SPEC CPU 2017).

Each execution generates very large trace files, routinely totalling many gigabytes, which must then be diffed. To achieve reasonable processing time, we picked a judicious subset of the available executions for each codebase—the aim being to cover as much code as possible, while keeping trace size and execution time manageable.

Using Git as an example, trace collection for a single variant (compiler version and optimisation level) produced ~259 GiB of traces and took 2–3 hours using 12 cores in parallel. Comparing two of these variants to find divergences took 3–4 hours using 24 cores in parallel. Our collection and analysis tools should be considered prototypes; we expect they could be made a good deal faster.

With the larger codebases, we picked from the test suite "breadth-first" across categories of tests, and also dropped any test witnessed to behave nondeterministically. Within SPEC CPU 2017, we selected those codebases written in C and/or C++ and not using `longjmp()` (incidentally not currently supported by the QBDI framework used by our prototype; this drops `perlbench` and `omnetpp`), and worked with the "rate" (not "speed") inputs for these codebases. With many of the SPEC benchmarks, we reduced the size of inputs and/or altered loop termination criteria, in order to explore as much code as possible (noting we are are not interested in the benchmark score). Tab. 3 details each package we examined and any systematic filtering we applied.

We compared the optimised traces to unoptimised ones using our comparison approach described earlier. In addition, we manually inspected portions of the traces to sanity-check the output and to look for interesting patterns.

Our tools detect the "expected" divergences of Tab. 1 and report a deduplicated count of affected source positions. They also detect remaining "unexpected" issues of Tab. 2; we report these to compiler authors for resolution; unlike "expected" issues, they can be fixed by adding or correcting debug info following existing formats and practices.

4.1 Measuring Divergence

We gathered call tree traces from unoptimised and optimised versions of executions of test suites from several codebases and established benchmarks. For each package, we ran either a systematically filtered portion of its test suite (tools) or a single input (benchmarks) as detailed in Tab. 3.

The following figures present divergence metrics by "bucket". A bucket is a group of the trace patterns (as in §3.3). All patterns in a bucket identify a common underlying compiler behaviour. The bucket names are shown in the legend of the plot.

Divergences are counted in terms of *normalised unique trace events*. Repeated instances of the same event are not counted twice. The normalised unique trace event fraction of a given bucket is the count of unique trace events in the diff that are matched by patterns in that bucket, divided by the count of unique trace events in entire unoptimised trace.

Coordinates changed divergences are further subdivided into "small" and "large" delta buckets by the size of the change. "Small" is a line delta of at most 3 lines and large is anything above that; this is arbitrary but works well in practice. Clang occasionally changes trace event coordinates to source line 0 (meaning no line), but with a non-0 column. We treat this as a variant of *coordinates removed* rather than doing arithmetic using zero.

4.2 Differences Across Compilers

Fig. 12 shows divergences found while executing Git compiled with Clang 13 and GCC 11 (released in 2021) as well as Clang 18 and GCC 14 (released in 2024). Both Clang and GCC display a trend

Table 3. Details of packages from which we collected and compared traces. We aimed to cover a reasonable portion of each package, while balancing constraints of trace size and execution time. Code size gives the coverable lines of code in the package. Bench coverage shows the coverage attained by the package's test suite or benchmark in its default configuration. Trace coverage shows the coverage reached during our experiments. Trace size gives the *mean* (not total) size of all traces produced by single variant (compiler family, compiler version, optimisation level) of that package. Individual tests producing very large traces (>1 GiB) were skipped.

Package	Code size (KLOC)	Bench cov. (KLOC)	Trace cov. (KLOC)	Trace size (MiB)	Notes
Git	180.3	144.3 (80%)	138.2 (77%)	265,329	Version control system, written in C
FFmpeg	541.7	284.1 (52%)	70.5 (13%)	14,848	Media transformer, written in C, selected one test file per feature group
SQLite	117.0	101.8 (87%)	74.4 (64%)	61,440	Database, written in C, selected one test file per feature group
SPEC CPU `502.gcc_r`	712.0	222.6 (31%)	66.0 (9%)	113	GNU C compiler, written in C, reduced test input
SPEC CPU `505.mcf_r`	1.5	1.2 (82%)	1.0 (64%)	0.2	Route planner, written in C, reduced test input
SPEC CPU `523.xalancbmk_r`	145.6	35.2 (24%)	30.2 (21%)	473	XML to HTML convertor via XSLT, written in C++, reduced test input
SPEC CPU `525.x264_r`	20.5	11.9 (58%)	6.2 (30%)	473	Video compressor, written in C, reduced resolution and frame count
SPEC CPU `531.deepsjeng_r`	6.1	4.9 (80%)	4.1 (67%)	104	Chess AI player, written in C++, reduced test input
SPEC CPU `541.leela_r`	4.4	2.4 (54%)	2.0 (46%)	494	Go AI player, written in C++, reduced test input, altered loop termination criteria
SPEC CPU `557.xz_r`	10.1	4.5 (45%)	3.3 (33%)	822	Data compressor, written in C, altered loop termination criteria

of more divergences in the newer version compared to the older version of the same compiler family. The distribution is a bit different, with GCC incurring far fewer divergences than Clang, and in particular, fewer inlining-related divergences. The divergence delta across Clang versions is much larger than for the GCC versions. This is partly explained by a known change to Clang's optimisations over time: Clang 15 enabled their tail-call transformation at O1 (instead of O2 as in earlier versions).[10] The increase in divergences can be interpreted as more optimisations that are not being compensated by debug info, i.e. the original call tree is recoverable less often.

4.3 Differences Across Optimisation Levels

Fig. 12 also shows divergences found while executing Git compiled with Clang 13 (released in 2021) at various optimisation levels. We see that divergences increase at higher optimisation levels. In Clang 13, the tail-call optimisation is first enabled at O2, and from manual review of the traces, this explains much of the additional divergences seen.

[10]There was some interest among LLVM developers, e.g. in bug D132623, in reverting the change in 2022 (which would move the tail-call optimisation back to O2), but this work stalled, so it remains enabled at O1 for now.

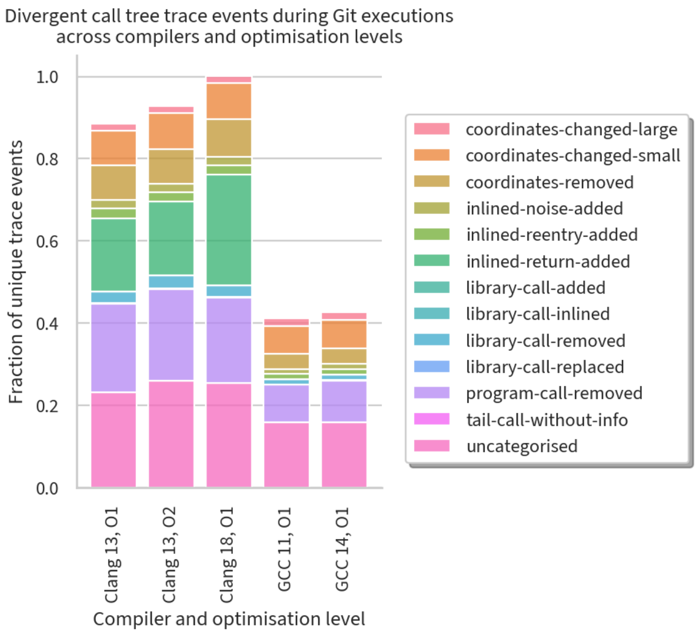

Fig. 12. Divergent call tree trace events seen from different compilers during executions of the Git codebase compiled at various optimisation levels.

GCC would be ideal to include in this comparison as well. Unfortunately, the GCC `O2` trace contained a preponderance (3%) of trace events for which no source coordinates could be found, and any remaining signal was lost in the large amount for diffing noise this caused. For all other compiler-level pairs, all trace events had at least function and file coordinates.

4.4 Differences Across Codebases

Fig. 13 shows divergences found while executing our test codebases compiled with Clang 18 at `O1`. With `505.mcf_r`, we see the smallest fraction of divergences. We suspect this is due to its relative simplicity compared to the other packages (see metrics in Tab. 3).

The various inlining-related buckets account for a sizeable portion of divergences. Manual inspection suggests quite a few of the remaining uncategorised divergences are also inlining-related. We were initially surprised by the size of the "program call removed" bucket. Many of these removed calls are to small utility functions which Clang either inlines (via the `always_inline` attribute) and reduces down to zero instructions or substitutes away using constraint propagation (SCCP) without describing them as inlined in debug info (see §3.4.2). We manually checked a few such cases to confirm that indeed, they are not described in debug info and debuggers cannot stop in these callees. Some traces show a number of artificial library calls added by the *optimiser* (instead of the frontend as we've mainly seen in this study). After manual inspection, the library call additions appear to be array initialisation loops that the optimiser transforms into `memset_pattern16` calls.

4.5 Compiler Bugs Found

Tab. 4 shows the compiler issues we have found and reported so far during this work. These were found by first running our automated divergence checking tool which highlights and summarises trace divergences. We then manually inspected a random sampling of these to better understand

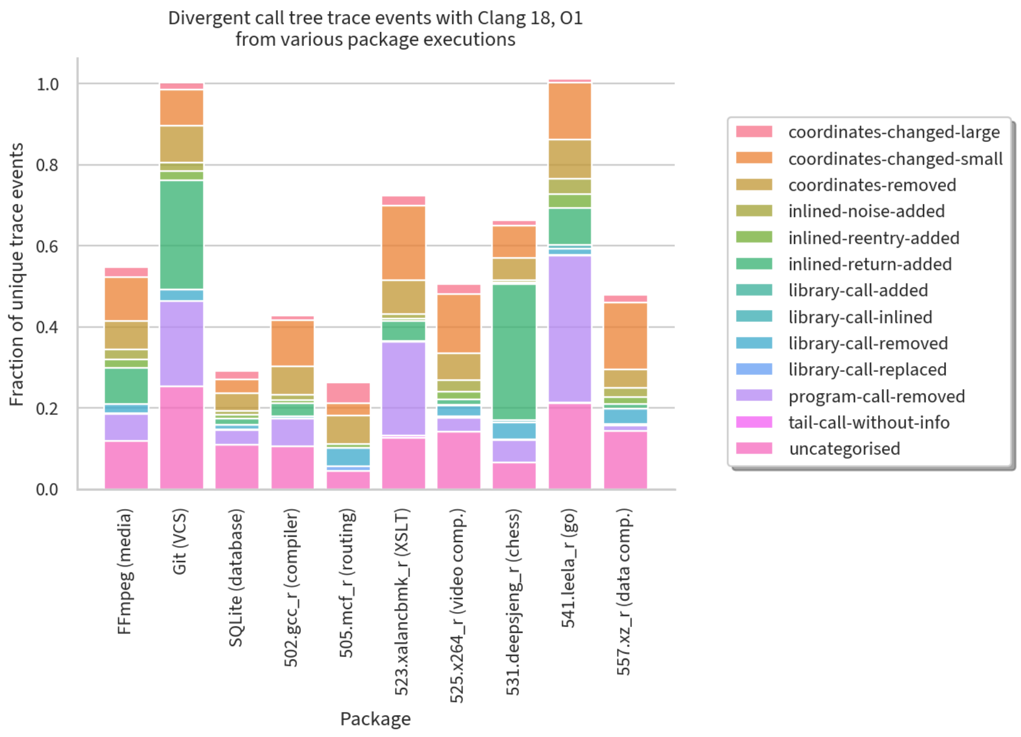

Fig. 13. Divergent call tree trace events during executions of various packages compiled with Clang 18 at O1.

Table 4. Details of compiler issues discovered. For each issue, we report its current status, e.g. whether it has been newly filed, confirmed by others, or fixed.

Project	ID	Summary	Status
LLVM	151185	Inlined subroutine should own call site entries, not parent subprogram	Confirmed
LLVM	156700	Inlined subroutine with loop misses part of PC range in debug info	Filed
LLVM	161962	Call site entries for declared C++ member functions not reported in debug info	Confirmed
GCC	121424	Debug info associates return instruction with inlined function	Confirmed

the root cause, which revealed the listed bugs. We believe it is likely there are more bugs yet to be revealed through continued application of this process.

LLVM issue 161962 shows that LLVM currently omits Dwarf call site info for declared C++ member functions. Our tool flagged this because it found that tail calls in packages including `523.xalancbmk_r` were not marked in debug info. This impact is broader than just the calls themselves: tail call info is part of "call site info" that also describes parameter values at function entry, and this information was omitted entirely, preventing the debugger also from showing these.

5 Related Work

5.1 Other Work on Debug Info Validation

Existing approaches to validating debugging information, such as Dexter [Bedwell 2018] used in LLVM, rely on manually annotating individual test cases with the expected state at each program point of interest. This can provide good test coverage, but at a high cost in engineer time.

Recent work has explored the correctness of Dwarf stack-unwinding (frame) info [Bastian et al. 2019], showing how to validate, synthesise, and also "compile" it (yielding a fast unwinder). This is separate from the information we consider.

The "actionable programs" technique [Li et al. 2020] works by generating many program variants, each adding (the effect of) `printf` for many combinations of variables and source coordinates, then automatically driving a debugger to check the values. While this approach can find *correctness* issues, it is a poor fit for checking *completeness* since `printf` inhibits optimisations that might remove variables' debug info. Another approach [Di Luna et al. 2021] collects "debugging traces" from random programs by single-stepping unoptimised and optimised binaries while performing sanity checking of line, frame, variable scope, and function argument values. These "trace invariants" can ensure metadata is plausible but not that it is actually correct.

A recent validation technique [Assaiante et al. 2023] proposes, in the absence of a test oracle, to instead find bugs more indirectly by searching for deviations from three empirical "conjectures" about what correct debug info is expected to look like, viz. that (1) locals should be available at external call sites passed as arguments, that (2) globals should be visible if they are written to, and that (3) locals' coverage should only decrease over a function. Conjecture 1 can be seen as codifying compilers' "preservation of input/output" contract as manifest at linkage (ABI) granularity; it is not clear it would be applicable for a whole-program optimiser, for example. The approach overall deliberately aims at a sanity check rather than a strong correctness criterion.

"Cross-level debugging" [Yang et al. 2025] is a bug-finding technique that compares debug-time behaviours seen by instruction-level stepping (e.g. `stepi` in GDB) with source-level (e.g. `step`). It shares with our work the theme of differential testing up to consistency properties. Its properties are that source-level stepping hits (only) a subset of the positions hit at instruction-level stepping, in the same order, and showing the same values for local variables. Unlike our work, this reveals bugs in debuggers, which perform stepping, but not the underlying debug info—which is identical across the compared "levels" and conceptually leaves no room for such inconsistencies. For example, in Dwarf the "subset" property is true by construction: the source positions marked `is_stmt` form a subset of all steppable positions, as a consequence of the line table format. Similarly, at a given object program state, the source position is identified in Dwarf by program counter and view number; given these, there is at most one possible valuation of each local variable.

"Robustifying" of line table information [Huang et al. 2025] detects cases where a single compiler pass's changes to line information would introduce spurious new source lines along any control-flow path. Given compiler passes manipulating IR using given primitives (Create, Clone, Move, and Replace) modelled on a subset of the LLVM API, it effectively computes afresh what the pass "should have" specified about the line information to avoid spurious results, flagging divergent cases as observed using an instrumented compiler. These divergences were found to be widespread in LLVM passes. However, passes using more general transformation primitives cannot be analysed in this way, and passes which restructure control flow are also largely out of scope.

5.2 Alternative Approaches to Debuggable Optimised Code

"Expected" versus "truthful". Prior work [Zellweger 1984] identified two categories of debugging experience maintained by the infrastructure: *truthful*, meaning the debugger shows states actually inhabited by the optimised program but translates them into source program terms as best it can, and *expected*, meaning the debugger's behaviour on an optimised programs is identical to the unoptimised case, i.e. consistent with a natural operational semantics of the source language. "Expected" semantics are typically offered by language virtual machines' debug servers, using either an interpreter or dynamic deoptimisation [Hölzle et al. 1992], while "truthful" approaches are often seen in compiler toolchains using debug info formats such as Dwarf [2017]. In the latter it is

considered a bug if compiling with -g changes what code is generated [Wang et al. 2023]. Other approaches often aim to get close to the level of optimisation allowed by truthful approaches using dynamic (re)compilation to inhibit optimisation only on demand (e.g. only when debugging is enabled at some point in the program) and minimally (still optimising around that point) [Van De Vanter et al. 2018].

"Expected" made faster. Several systems perform the feat of switching to unoptimised code, requiring *on-stack replacement* if the switch is made during a live function call [D'Elia and Demetrescu 2018; Guo 2014; Hölzle et al. 1992; Zurawski and Johnson 1991]. These systems generally exist within just-in-time compilers, and rely on the compiler to annotate specific program points where it is safe to jump from optimised to unoptimised mode for debugging, and avoid some optimisations across such points. Such compilers are usually co-designed with garbage collectors relying on similar special "safepoints", i.e. points where complete pointer maps are available.

"Truthful" fidelity improvements. By contrast, "truthful" approaches treat the machine program state as the ground truth, but try to lift this up to a source-level view where possible; this will not slow down execution, but risks limited coverage. The earliest research into debugging optimised programs addresses improving this approach, by propagating mapping information through the passes of the compiler [Coutant et al. 1988; Hennessy 1982]. Subsequent techniques for improving this include the value expressions of Dwarf [2017] and similar formats.

Hybrid approaches. [Zellweger 1984] describes how to compare unoptimised and optimised program data-flow and control-flow to untangle state changes of specific optimisations and recover variable values. OPTVIEW [Tice and Graham 1998] adds to the pure "truthful" a "modified" source listing that models the optimised program and so may not match the original source.

Omniscience or "knowledge levels". Tice [1999] implemented a state-residualisation technique for local variables in the Optdbx debugger, remembering "dead" values while they are still in scope but no longer register-allocated. *Omniscient debuggers* such as Pernosco, built on record/replay systems such as `rr` [O'Callahan et al. 2017], 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. As a result, these techniques are less suited to in-the-field debugging and are arguably less "truthful" than a conventional debugger such as gdb over `ptrace`, which does not perform such narrowing. These tools' user experience still suffers whenever there are gaps in the coverage or correctness of Dwarf or its equivalents, of the kind we address in this work.

6 Conclusions and Future Work

Our exploratory work appears to show that in the context of dynamic call tree recovery, there are relatively few *unexpected* divergences that compiler authors may be able fix relatively easily, at least at optimisation settings that exclude inlining. Most trace divergences are in fact the more complex *expected* variety, which may need debug info format extensions (or at least compiler author policy changes) to resolve. We view this as a hopeful outcome: new debug info features can be designed to address expressiveness gaps, and compilers can then improve their output incrementally as those new features are implemented. The ability to recover dynamic call trees sets a strong correctness criterion, albeit only for this particular "control-centric" dimension of debug info; we hope similarly strong correctness criteria for other aspects can be explored in future work.

Data-Availability Statement

We have published an open-source artifact [Stinnett and Kell 2026] containing both the tools we developed for this research and the experimental pipelines used to acquire, process, and display our results.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) via grant EP/W012308/1 and the Defense Advanced Research Projects Agency (DARPA) under contract HR001124C0488.

References

Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proc. of PLDI '97*. doi:10.1145/258915.258924

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*. doi:10.1145/3575693.3575720

Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli. 2019. Reliable and Fast DWARF-based Stack Unwinding. *PACMPL* 3, OOPSLA (Oct. 2019), 146:1–146:24. doi:10.1145/3360572

Greg Bedwell. 2018. Measuring the User Debugging Experience. Presented at European LLVM Developers Meeting. Retrieved 2022-04-25 from https://llvm.org/devmtg/2018-04/slides/Bedwell-Measuring_the_User_Debugging_Experience.pdf

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

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.* 25, 2 (June 1996), 493–504. doi:10.1145/235968.233366

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*. doi:10.1145/53990.54003

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

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*. doi:10.1145/3445814.3446695

DWARF Debugging Information Format Committee. 2017. DWARF Debugging Information Format: Version 5. Retrieved 2019-07-29 from https://dwarfstd.org/doc/DWARF5.pdf

Shu-Yu Guo. 2014. Debugging in the Time of JITs. Retrieved 2023-02-22 from https://rfrn.org/~shu/2014/05/14/debugging-in-the-time-of-jits.html

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

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

Shan Huang, Jingjing Liang, Ting Su, and Qirun Zhang. 2025. Robustifying Debug Information Updates in LLVM via Control-Flow Conformance Analysis. *Proc. ACM Program. Lang.* 9, PLDI, Article 168 (June 2025). doi:10.1145/3729267

Charles Hubain and Cédric Tessier. 2017. Implementing an LLVM Based Dynamic Binary Instrumentation Framework. Retrieved 2025-03-07 from https://media.ccc.de/v/34c3-9006-implementing_an_llvm_based_dynamic_binary_instrumentation_framework

Jakub Jelínek. 2010. Improving Debug Info for Optimized Away Parameters. In *GCC Summit*. Retrieved 2021-11-10 from https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=jelinek.pdf

Stephen Kell and J. Ryan Stinnett. 2024. Source-Level Debugging of Compiler-Optimised Code: Ill-Posed, but Not Impossible. In *Proc. of Onward! '24*. doi:10.1145/3689492.3690047

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

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*. Retrieved 2023-04-22 from https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

Alexandre Oliva. 2017. Statement Frontier Notes and Location Views. Retrieved 2022-09-08 from https://developers.redhat.com/blog/2017/07/11/statement-frontier-notes-and-location-views

Alexandre Oliva. 2019. GCC gOlogy: Studying the Impact of Optimizations on Debugging. Retrieved 2022-05-25 from https://www.fsfla.org/~lxoliva/writeups/gOlogy/gOlogy.html

J.E. Smith and A.R. Pleszkun. 1988. Implementing Precise Interrupts in Pipelined Processors. *IEEE Trans. Comput.* 37, 5 (May 1988), 562–573. doi:10.1109/12.4607

J. Ryan Stinnett and Stephen Kell. 2024. Accurate Coverage Metrics for Compiler-Generated Debugging Information. In *Proc. of CC '24*. doi:10.1145/3640537.3641578

J. Ryan Stinnett and Stephen Kell. 2026. Debugging Debugging Information using Dynamic Call Trees (artifact). doi:10.5281/zenodo.18392053

Caroline Tice. 1999. *Non-Transparent Debugging of Optimized Code.* Ph. D. Dissertation. University of California, Berkeley. https://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-99-1077.pdf

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

Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and Other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (March 2018). doi:10.22152/programming-journal.org/2018/2/14

Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. 2023. Compilation Consistency Modulo Debug Information. In *Proc. of ASPLOS '23*. doi:10.1145/3575693.3575740

Yibiao Yang, Maolin Sun, Jiangchang Wu, Qingyang Li, and Yuming Zhou. 2025. Debugger Toolchain Validation via Cross-Level Debugging. In *Proc. of ASPLOS '25*. doi:10.1145/3669940.3707271

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

Polle T. Zellweger. 1984. *Interactive Source-Level Debugging of Optimized Programs.* Ph. D. Dissertation. University of California, Berkeley. Retrieved 2022-09-25 from https://search.library.berkeley.edu/permalink/01UCS_BER/1thfj9n/alma991002570669706532

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

Received 2025-10-10; accepted 2026-02-17